AFRL-RI-RS-TR-2014-269

AUTOMATIC REQUIREMENTS SPECIFICATION EXTRACTION FROM NATURAL LANGUAGE (ARSENAL)

SRI INTERNATIONAL

OCTOBER 2014

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2014-269 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /
STEVEN L. DRAGER
Work Unit Manager

/ S /
MARK H. LINDERMAN
Technical Advisor, Computing
& Communications Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* OCT 2014	**2. REPORT TYPE** FINAL TECHNICAL REPORT	**3. DATES COVERED** *(From - To)* SEP 2012 - MAR 2014

4. TITLE AND SUBTITLE

AUTOMATIC REQUIREMENTS SPECIFICATION EXTRACTION FROM NATURAL LANGUAGE (ARSENAL)

5a. CONTRACT NUMBER
FA8750-12-C-0339

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62303E

6. AUTHOR(S)

Shalini Ghosh, Natarajan Shankar, Patrick Lincoln, Daniel Elenius, Wenchao Li, Wilfrid Steiener

5d. PROJECT NUMBER
HACM

5e. TASK NUMBER
SL

5f. WORK UNIT NUMBER
NL

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94555

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2014-269

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. PA# 88ABW-2014-4836 Date Cleared: 21 October 2014

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Natural language (supplemented with diagrams and some mathematical notations) is convenient for succinct communication of technical descriptions between the various stakeholders (e.g., customers, designers, implementers) involved in the design of software systems. However, natural language descriptions can be informal, incomplete, imprecise and ambiguous, and cannot be processed easily by design and analysis tools. Formal languages, on the other hand, formulate design requirements in a precise and unambiguous mathematical notation, but are more difficult to master and use. We propose a methodology for connecting semi-formal requirements with formal descriptions through an intermediate representation. We have implemented this methodology in a research prototype called Automatic Requirements Specification Extraction from Natural Language (ARSENAL). The main novelty of ARSENAL lies in its ability to generate a fully-specified complete formal model automatically from natural language requirements. ARSENAL extracts relations from text using semantic parsing and progressively refines them over multiple stages to create a final composite model. Currently, ARSENAL generates formal models in linear-time temporal logic (LTL), but the approach can be adapted for other models, e.g., probabilistic relational models like Markov Logic Networks (MLN). The formal models of the requirements can be used to check important design and system properties, e.g., consistency, satisfiability, realizability. ARSENAL has a modular and flexible architecture that facilitates porting it from one domain to another. We evaluated ARSENAL on complex requirements from two real-world case studies: the Time-Triggered Ethernet (TTEthernet) communication platform used in space, and FAA-Isolette infant incubators used in NICU. We systematically evaluated various aspects of ARSENAL — the accuracy of the natural language processing stage, the degree of automation, and robustness to noise.

15. SUBJECT TERMS

Requirements, natural language processing, temporal logic, model synthesis, formal verification

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** **STEVEN L. DRAGER**
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	44	**19b. TELEPHONE NUMBER (*Include area code*)** **(315) 330-2735**

Table of Contents

List of Figures

List of Tables

1 Summary

Natural language (supplemented with diagrams and some mathematical notations) is convenient for succinct communication of technical descriptions between the various stakeholders (e.g., customers, designers, implementers) involved in the design of software systems. However, natural language descriptions can be informal, incomplete, imprecise and ambiguous, and cannot be processed easily by design and analysis tools. Formal languages, on the other hand, formulate design requirements in a precise and unambiguous mathematical notation, but are more difficult to master and use. We propose a methodology for connecting semi-formal requirements with formal descriptions through an intermediate representation. We have implemented this methodology in a research prototype called Automatic Requirements Specification Extraction from Natural Language (ARSENAL). The main novelty of ARSENAL lies in its ability to generate a fully-specified complete formal model automatically from natural language requirements. ARSENAL extracts relations from text using semantic parsing and progressively refines them over multiple stages to create a final composite model. Currently, ARSENAL generates formal models in linear-time temporal logic (LTL), but the approach can be adapted for other models, e.g., probabilistic relational models like Markov Logic Networks (MLN). The formal models of the requirements can be used to check important design and system properties, e.g., consistency, satisfiability, realizability. ARSENAL has a modular and flexible architecture that facilitates porting it from one domain to another. We evaluated ARSENAL on complex requirements from two real-world case studies: the Time-Triggered Ethernet (TTEthernet) communication platform used in space, and FAA-Isolette infant incubators used in Neonatal Intensive Care Units (NICUs). We systematically evaluated various aspects of ARSENAL — the accuracy of the natural language processing stage, the degree of automation, and robustness to noise.

2 Introduction

Software systems operate in the real world, and often work in conjunction with complex physical systems. Many different stakeholders participate in the design and operation of these systems. Requirements specify important properties of (cyber-physical) software systems, e.g., conditions required to achieve an objective, or desired invariants of the system. Requirements in formal languages are precise and complete – the formal models for requirements are useful for checking consistency and verifying properties, but are cumbersome to specify. As a result, stakeholders (e.g., customers, designers, engineers) often prefer writing requirements in natural language (NL). The NL requirements can be written easily without burden of formal rigor, but can be imprecise, incomplete, and ambiguous.

So, natural language descriptions and formal modeling languages each offer distinct advantages to the system designer. The informality of natural language can kick-start discussion among stakeholders in early design, but can lead to confusion, lack of automation, and errors. The rigor of formal languages can eliminate broad classes of ambiguity, enable consistency checking, and facilitate automatic test case generation — However, mastery of formal notations requires a significant amount of training and mathematical sophistication.

Another important issue to consider is the cost of errors — most of the costly errors often enter at the requirements stage as a result of confusion among stakeholders [BP98]: “If a defect is found in the requirements phase, it may cost $1 to fix. It is proffered that the same defect will cost $10 if found in design, $100 during coding, $1000 during testing [bug10].” In order to catch as many errors as possible during the requirements phase, iterations between the stakeholders through clear communication in natural language must be supported. Formal models and descriptions that can detect errors, incompleteness, ambiguity, and inconsistency in the requirements should also be used. By bridging the gap between semi-formal requirements and formal specifications, we can dramatically reduce the number of costly uncaught errors in requirements and enable high levels of assurance for critical complex systems. Figure 1 summarizes the tradeoff between natural language and formal requirements specifications.

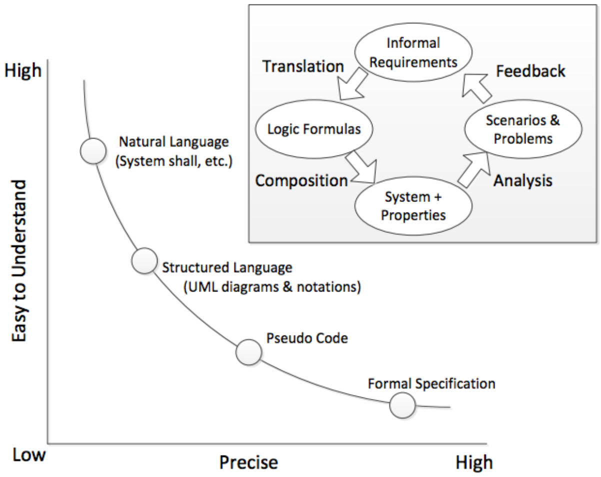


Figure 1 Tradeoff between natural language and formal specifications [Bab07], inset showing the design-iteration cycle of the ARSENAL methodology.

We aim to leverage the best of both natural and formal languages to aid the system designer in achieving high assurance for critical systems. This primary objective of this Final Technical Report (FTR) is to answer this question:

Can we build a requirements engineering framework that combines the strengths of semi-formal natural language and precise formal notations?

To that effect, we present the ARSENAL methodology. ARSENAL uses state-of-the-art advances in natural language processing (NLP) and formal methods (FM) to connect natural language descriptions with their precise formal representations. ARSENAL provides a method for extracting relevant information from NL requirements documents and creating formal models with that information — it is an exploratory and experimental open-loop framework for extracting formal meaning from semi-formal text.

Let us consider the following sentence, which is part of the requirements specification for a regulator that regulates the temperature in an isolette (an incubator for an infant that provides controlled temperature, humidity, and oxygen):

REQ1: *If the Status attribute of the Lower Desired Temperature or the Upper Desired Temperature equals Invalid, the Regulator Interface Failure shall be set to True.*

This requirements sentence is written with terminology that is particular to the domain, in a language that facilitates comprehensible communication between the relevant stakeholders involved in different stages of the isolette design. ARSENAL aims to convert such requirements automatically to formal models, providing a natural language front-end to formal analysis that is robust and flexible across different forms of natural language expressions in different domains, and customized to stylized usages within each domain.

The input of ARSENAL consists of requirements in natural language with specific technical content. ARSENAL uses domain-specific semantic parsing to extract relations from text, and progressively refines them over multiple stages to create formulas in first-order logic (FOL) or LTL. These are then used to create a composite model in the FM stage, which can be used by formal verification tools like theorem provers (e.g., Prototype Verification System (PVS) [ORR$^+$96]) and model checking tools (e.g., SAL [BGL$^+$00]) as well as LTL synthesis tools (e.g. RATSY [BCG$^+$10]) for automated analysis of the formal specifications. The results provide concrete empirical evidence that it is possible to bridge the gap between natural language requirements and formal specifications, achieving a promising level of performance and domain independence.

The main challenges and key insights of ARSENAL are outlined in Table 1.

Table 1 Key innovations in ARSENAL.

	Challenges	**Key Insights**
1	Bridge the gap between semi-formal natural language requirements and precise formal models.	Create a rich/expressive intermediate representation (IR), useful for generating outputs in multiple formalisms.
2	Create a general-purpose architecture that can be ported to different domains.	Encapsulate domain-specific components in modules (e.g., NL preprocessor, output generators), keeping rest of system domain-independent and reusable.
3	Incorporate semantics into formal model generation.	Add semantics to the formal model via rewrite rules and type inference algorithms in the model generator stage.

The organization of the rest of the FTR is as follows: Section 3 gives some relevant background, Section 4 gives an overview of ARSENAL, while Sections 5 and 6 respectively describe the NLP and FM stages in more detail. Section 7 discusses the results of our experiments with ARSENAL on the FAA-Isolette and TTEthernet requirements documents, while Section 8 discusses the novelty of ARSENAL compared to related research. Finally, Section 9 summarizes the contributions of this work and outlines possible future directions of research.

3 Background

3.1 The SAL Modeling Language

In this FTR, we focus on transition systems modeled using the Symbolic Analysis Laboratory (SAL) language [BGL+00]. At the core, SAL is a language for specifying transition systems in a compositional way. A transition system is composed (synchronously or asynchronously) of modules, where each module consists of a *state type*, an *invariant definition* on this state type, an *initialization condition* on the state type, and a *binary transition relation* on the state type. The state type is defined by four pairwise disjoint sets of *input*, *output*, *local* and *global* variables. Both *input* and *global* variables are observed variables of a module. *Output*, *global* and *local* variables are *controlled* variables of a module. In this FTR, we additionally distinguish between which controlled variables are *state* variables and which are *logical* variables (variables which do not directly define the state space).

The semantics of a SAL transition system is given by a Kripke structure. A Kripke structure over a set of atomic propositions AP is a tuple $\langle Q, Q_0, R, L\rangle$, where Q is the set of states, $Q_0 \subseteq Q$ is the set of initial states, $R \subseteq Q \times Q$ is the transition relation, and $L : Q \to 2^{AP}$ is a labeling function that assigns each state to a set of AP that is true in that state. Kripke structure is a kind of nondeterministic automaton that is widely used in model checking [CGP99].

An example of a SAL program is given in Figure 2. This model represents a transition system whose state space is defined by the local variable "count". Whenever input "add" is `true` , "count" is incremented by 1 (the apostrophe on "count" indicates it is the next state of "count"), otherwise, its value remains unchanged. This model also has a single output "out" having a record type. If "count" is greater than 100, the "high" field is set to `true` , and if "count" is less than 50, the "low" field is set to `true` . In this example, "main" is the name of a module. For simplicity, we consider only self-contained modules in this FTR. "Type1 ..." is a type expression that creates "Type1" as a record type. In Section 5, we will describe how the type information is automatically inferred by ARSENAL. The other relevant SAL language constructs for defining a transition system are "DEFINITION", "INITIALIZATION" and "TRANSITION". The "DEFINITION" section describes constraints over the *wires* of a module. The "INITIALIZATION" section gives the initial conditions of the *controlled* variables. Finally, the "TRANSITION" section constrains the possible next states for the *state* variables. In this example, the update of "count" is given by the guarded command with the guard "add = TRUE" and the command "count' = count + 1", where the apostrophe indicating it is referring to the next state of "count". We refer the readers to [BGL+00] for a detailed description of the language. In general, SAL models are expressive enough to capture the transition semantics of a wide variety of source

```
sal_example : CONTEXT =
BEGIN
  Type1 : TYPE = [# high : BOOLEAN, low : BOOLEAN #];
  main : MODULE =
  BEGIN
    INPUT add: BOOLEAN
    LOCAL count: INTEGER
    OUTPUT out Type1
    DEFINITION
      out.high = count > 100;
      out.low = count < 50;
    INITIALIZATION
      count = 0;
    TRANSITION
    [ add = TRUE --> count’ = count + 1
      []
      ELSE --> count’ = count ]
  END;
  THEOREM main |- G (NOT(out.high = TRUE));
END
```

Figure 2 Example SAL Model.

languages. One key contribution of ARSENAL is an automatic way of translating from the NL description of a transition system and its requirements directly to a SAL model and theorems.

3.2 Verification and Synthesis with LTL

In addition to specifying a transition system, a sentence in NL may be describing a high-level requirement that the system has to satisfy. Often times, a requirement can be precisely captured using temporal logic. In this FTR we use LTL [MP92], whose semantics can be interpreted over Kripke structures. For example, in Figure 2, the SAL theorem states that globally (for any computation), "high" is never `true`. We consider two problems that leverages LTL to reason about potential inconsistencies in a NL documentation, namely, verification and synthesis.

For verification, we use model checking to analyze if a SAL model satisfies its LTL specification. In general, the application of model-checking tools involves a nontrivial step of creating a mathematical model M of the system and translating the desired properties into a formal specification ψ. ARSENAL's NLP stage automates this process of creating M and ψ with minimal user guidance. Given a model M as a (labeled) transition system and a specification ψ in LTL, both produced in the NLP stage of ARSENAL, we check if $M \models \psi$, that is, if the model satisfies the specification. When the model does not satisfy the specification, a negative answer, often in the form of a counterexample, is presented to the user as an certificate of how the system fails the specification. In this FTR, we use SAL's bounded model checking [CBRZ01] capability as the main workhorse for finding such inconsistencies.

ARSENAL also provides the flexibility of generating only specifications from the natural language requirement. Consistency in this case means if the specification is *satisfiable*, that is, whether a model exists for the specification. This problem is known as LTL satisfiability checking and it can be reduced to model checking [RV07]. Given a LTL formula ψ, it is satisfiable precisely when an *universal* model M[1] does not satisfy $\neg\psi$. Similarly, a counterexample that points to the inconsistency is produced when ψ is not satisfiable.

Given a LTL specification, it may also be possible to directly *synthesize* an implementation that satisfies the specification. Satisfiability may seem as a sufficient condition for a design specification to be implemented. However, it is known that a stronger condition, known as *realizability*, precisely defines when a specification can be realized. We use a technique called *LTL synthesis* to reason about whether such a target implementation is *realizable* given the generated specification. *Realizability*, the decision problem of determining whether such an implementation exists, can be used to further inspect the requirement document for inconsistencies, with ARSENAL facilitating the connection between NL requirement and logical specification. If the specification is realizable, then a Moore machine can be extracted as an implementation that satisfies the specification. Thus, the benefit of LTL synthesis is a correct-by-construction process that can automatically generate an implementation from its specification. In general, LTL synthesis has high computational complexity [Kup12]. However, it has been shown that a subclass of LTL, known as Generalized Reactivity (1) (GR(1)), is more amenable to synthesis [PP06] and is also expressive enough for specifying complex industrial designs [BGJ$^+$07]. The details of the LTL satisfiability and synthesis approaches we took are discussed in Section 6.

4 Methods, Assumptions, and Procedures

In this section, we give an overview of the flow (shown in Figure 3) using the example requirement sentence REQ1 from the FAA-Isolette corpus, introduced in Section 2.

Given requirements written in natural language, ARSENAL first processes them using a NLP stage. The NLP stage has a preprocessor that does some domain-independent processing (e.g., identifying arithmetic formulas) as well as domain-specific processing (e.g., identifying domain-specific nominal phrases corresponding to an entity). In REQ1, the preprocessor identifies terms like *Lower Desired Temperature*, *Upper Desired Temperature* and *Regulator Interface Failure* as phrases with special meanings in the FAA-Isolette

[1] Universal means M contains all possible traces over the set of atomic propositions.

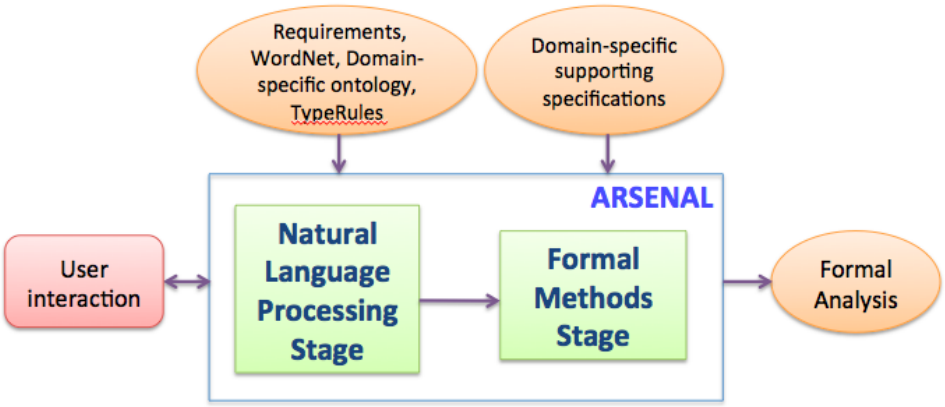

Requirements, WordNet, Domain-specific ontology, TypeRules
Domain-specific supporting specifications
ARSENAL
User interaction
Natural Language Processing Stage
Formal Methods Stage
Formal Analysis

Figure 3 ARSENAL pipeline.

domain, and converts each of these phrases to a single term (corresponding to an entity in this domain). This results in the following preprocessed requirements sentence:

If the Status_attribute of the Lower_Desired_Temperature or the Upper_Desired_Temperature equals Invalid, the Regulator_Interface_Failure shall be set to True.

The output of the preprocessor is analyzed by a semantic processor that first does shallow semantic parsing of the preprocessed requirements text using the Stanford Type Dependency Parser (STDP) [dMMM06]. STDP outputs typed dependencies such as: `nsubj(equals, Status_attribute)`.

Each typed dependency indicates a semantic relation between parsed terms, e.g., the typed dependency above indicates that `Status_attribute` is a subject of the verb `equals`. The next stage of the semantic processor converts these typed dependencies to entries in a symbol table in an intermediate representation (IR) of the form:

```
Upper_Desired_Temperature-9: Upper_Desired_Temperature | entity | unique | or:
[Lower_Desired_Temperature-6]
```

The IR table maps each symbol to its metadata and to its relationships with other symbols. In the example above, the IR table entry for `Upper_Desired_Temperature` shows that it is an entity, it is unique, and is connected to another entity `Lower_Desired_Temperature` via the relation `or`. A detailed description of the IR table is given in Section 5.

The next part of the ARSENAL pipeline is the FM stage, which converts the IR table to a formal model (in the current ARSENAL prototype, we generate a SAL model). ARSENAL effectively converts multiple NL requirements sentences, which describe a system module, into a unified SAL model. Using this model, we can potentially generate a proof or counter-example for certain system properties of interest.

Note that ARSENAL is a general purpose methodology. We can plug in different modules to various parts of the workflow, e.g., any state-of-the-art typed dependency parser in the NLP stage or formal analysis tool in the FM stage. In this instance of the ARSENAL pipeline, we use STDP and SAL in the NLP stage and FM stage respectively (as described in the following sections), but other tools can also be plugged into these stages.

5 Natural Language Processing

The NLP stage takes requirements in natural language as input and generates the IR table as output. The different components of the NLP stage (shown in Figure 4) are described in detail in this section.

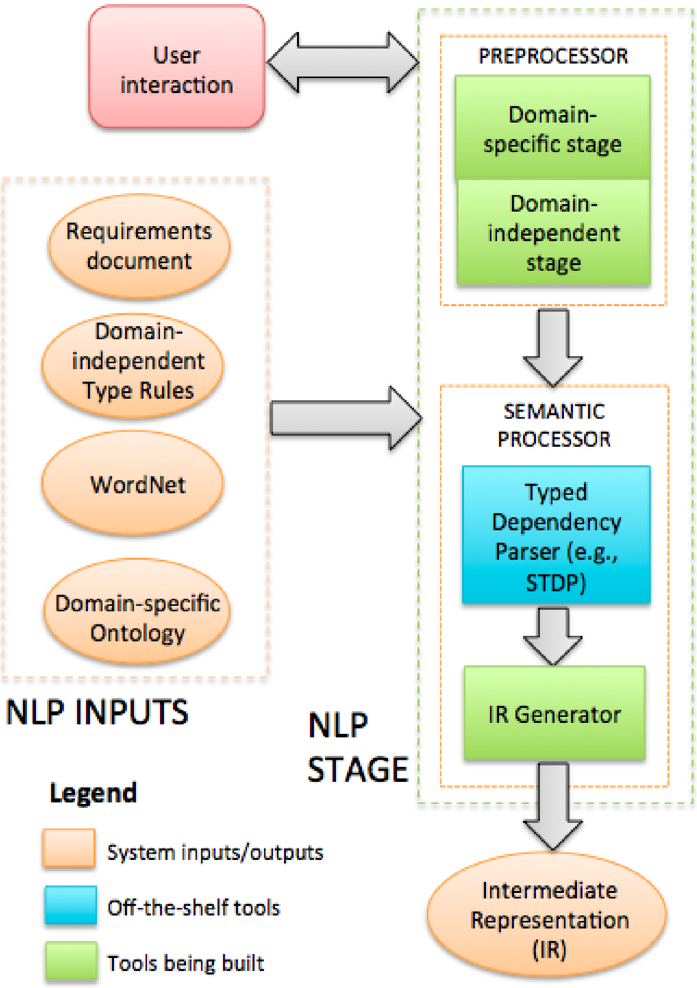


Figure 4 NLP Stage of ARSENAL pipeline.

5.1 Preprocessor

The first part of the NLP stage is a preprocessor. It seeks to extract better (more meaningful) parses to aid the Stanford parser using both domain-specific and domain-independent transformations on the requirements sentence. An example domain-specific preprocessing task is identifying entity phrases like "Lower Desired Temperature" and converting them to the term `Lower_Desired_Temperature`. Domain-independent preprocessing tasks include identifying and transforming arithmetic expressions, so that NLP parsers like the Stanford parser can handle them better. For example, the preprocessor replaces the arithmetic expression “[x + 5]” by `ARITH_x_PLUS_5`. The parser then treats this as a single term, instead of trying to parse the five symbols in the arithmetic expression. The preprocessor also encodes complex phrases like “is greater than or equal to” into simpler terms like `dominates`. In later processing (e.g., in the Formal Methods stage), ARSENAL decodes the original arithmetic expressions from the corresponding encoded strings. Apart from detecting n-grams and converting them to phrases (based on the input glossary), and doing arithmetic processing, the pre-processor also inserts missing punctuation (e.g., commas at phrase boundaries) and does other text transforms — these help the downstream parser get better results.

5.2 Stanford Typed Dependency Parser

The next part of the NLP stage is the application of the STDP to the preprocessed sentence. The syntactic parser in STDP parses the requirements text to get unique entities called mentions, while the dependency parser detects grammatical relations between the mentions. The final output is a set of typed dependency (TD) triples between extracted terms, which encode the grammatical relationship between mentions extracted from a sentence. For the example requirement REQ1, the full set of TDs generated by STDP are shown in Figure 5. The Stanford typed dependencies representation provides a simple description of the grammatical relationships in a sentence, which can be understood easily and used effectively to extract textual relations without requiring deep linguistic expertise [dMMM06].

```
mark(equals-10, If-1)
det(Status_attribute-3, the-2)
nsubj(equals-10, Status_attribute-3)
det(Lower_Desired_Temperature-6, the-5)
prep_of(Status_attribute-3, Lower_Desired_Temperature-6)
det(Upper_Desired_Temperature-9, the-8)
prep_of(Status_attribute-3, Upper_Desired_Temperature-9)
conj_or(Lower_Desired_Temperature-6, Upper_Desired_Temperature-9)
advcl(set-17, equals-10)
dobj(equals-10, Invalid-11)
det(Regulator_Interface_Failure-14, the-13)
nsubjpass(set-17, Regulator_Interface_Failure-14)
aux(set-17, shall-15)
auxpass(set-17, be-16)
root(ROOT-0, set-17)
prep_to(set-17, True-19)
```

Figure 5 STDP output for REQ1.

Note that the suffix of each mention is a number indicating the word position of the mention in the sentence, e.g., `equals-10` refers to the mention `equals` at position 10 in the sentence. The position index helps to uniquely identify the mention if it has multiple occurrences in the sentence. The TDs output by STDP are triples of the form:

```
<relation name> (<governor term>, <dependent term>)
```

Each triple indicates a relation of type “relation name” between the governor and dependent terms. For example, let us consider the TD:

```
prep_of(Status_attribute-3, Lower_Desired_Temperature-6)
```

It indicates that the mention `Status_attribute` is related to `Lower_Desired_Temperature` via a prepositional connective of type “of”.

We will denote governer terms by `?g` and dependent terms by `?d`.

5.3 Semantic Processor

The semantic processor starts with the STDP output and creates a mention table, by selecting each mention in STDP output and creating a hash from each mention to all TDs it is involved in. Subsequently, it uses the mention table to create the IR table, using Metadata tags and TypeRules. The overall algorithm for generating the IR is outlined in Figure 6.

Input: Requirements text, WordNet, Domain-specific ontology, TypeRules, Pre-processing rules.

Output: Intermediate Representation (IR) table.

1. **Run the requirement text through Stanford Dependency Parser:** This produces a graph of Typed Dependencies (TDs), and part of speech (POS) tags for all the words in the sentence.
2. **Create a Mention Table:** Select each MentionId in Stanford Parser output and creating a hash from MentionId to all typed dependencies (TDs) it is involved in.
3. **Initialize IR:** Empty at beginning.
4. **Populate the IR:** Iterate over MentionIds in the Mention Table in sequence. For each MentionId:
   (a) Get the POS tag for the MentionId.
   (b) Set *word* to the stem of the word (from the WordNet stemmer) and add an IR entry for the word.
   (c) If word is a math expression encoded by the math preprocessor, set its IR type to *arithmetic*.
   (d) Else if word is marked as a unique entity in the ontology, set its IR type to entity and its quantifier to *unique*.
   (e) Else if word is marked as predicate in the ontology, set its IR type to *pred*.
   (f) Else if word is a number, set its IR type to *num*.
   (g) Else if word has a noun POS tag, set its IR type to *entity*. In addition, if the word is not found in WordNet, set its quantifier to *unique* (as it is presumably a proper name).
   (h) Else if word has a verb POS tag, set its type to *event*.
5. **Execute the type rules by doing a DFS traversal of the TD graph:** For each MentionId in the Mention Table, for each TD associated with MentionId in the Mention Table, for each type rule TR:
   (a) Match the type rule with the TD, producing TD.
   (b) If step 5(a) was successful, i.e., the the left-hand-side of TR matches TD, execute the right-hand-side of TD.

Figure 6 Detailed algorithmic flow of IR generation.

Metadata tags Different types of metadata tags annotate the entries in the IR table:

1. TermType: Whether term is of the type entity, event, numeric, or predicate.
2. NegatedOrNot: Whether term is logically negated.
3. QuantifierType: Unique, all, exists.

4. Relations/Attributes: Temporal or normal.
5. Lists: Corresponds to the connectives and, or, implied-by.

These tags are used to associate semantics with the entries in the IR table. The metatag annotations are similar to role annotations in automatic semantic role labeling [GJ01]. In this stage, ARSENAL uses WordNet [Mil95] to identify word stems, and to find out whether a term is a known noun — if the term is a noun but not found in WordNet, it is marked as unique since it is most likely a proper noun. ARSENAL can also use special domain-specific ontologies or glossaries in this stage to annotate the IR entries with a richer set of metadata tags that can be used in downstream processing.

TypeRules TypeRules are domain-independent semantic rules used by the semantic processor to create the IR table. For example, `nsubjpass(V, N)` in the STDP output indicates that the noun phrase `N` is the syntactic subject of a passive clause with the root verb `V` . The TypeRule corresponding to the TD `nsubjpass(V, N)` indicates that `N` is a who/what argument of relation `V` in the output formula. TypeRules have the form:

```
TD(ARG1,ARG2) : ACTION(ARG3,ARG4)
```

For example:

```
prep_upon(?g,?d) : implies(?d,?g)
```

Matching of Typed Dependencies with TypeRules: Matching this rule with the TD: `prep_upon(entering-17, set-4)` produces a match with `?g = entering-17, ?d = set-4`. The action to execute is then: `implies(set-4, entering-17)`.

There are a number of different types of actions, each with its own semantics. The `implies(?x,?y)` action adds an entry `impliedBy:?x` to the IR entry for `?y`.

ARSENAL has different kinds of TypeRules, e.g., for handling implications, conjunctions/disjunctions, universal/existential quantifiers, temporal attributes/relations, relation arguments, and events.[2]

Rules with complex patterns: Some TypeRules are complex and have multiple TDs that match on the left-hand side. For example:

```
advcl(?g,?d) & mark(?d,if) : implies(?d,?g)
```

Here, the current TD being matched must match the first part of the rule, `advcl(?g,?d)`, but any other TD from the parsed sentence can be used to match the rest of the rule, i.e., `mark(?d,if)`. Note that "if" here is not a variable: it denotes that a mention of the word `if` must appear in that position in a TD to produce a match.

TypeRules can also contain an additional condition on the left-hand side. For example:

```
nsubj(?g,?d) & event(?g) : rel(agent,?g,?d)
```

Here, we have an additional check that whatever mention matches `?g` is marked as an event in the IR (in step 4 of the algorithm in Figure 6) — `rel(agent,?g,?d)` adds a relation `agent=?d` to the IR entry for `?g`.

A plug-and-play architecture like ARSENAL has certain flexibilities that enables it to give better performance by increasing accuracy of requirements processing. For example, if the shallow semantic NL parser generates multiple candidate parses of the requirements, ARSENAL can use semantic rules to select the best parse. ARSENAL can also correct inconsistencies between different NLP modules if they give conflicting results.

Figure 7 shows the full IR table corresponding to REQ1.

5.4 From Dependency Graph to Predicate Graph

One key challenge we face in ARSENAL is the mapping from natural language sentences to LTL formulas in a way that the semantics (according to LTL) are correctly preserved. Specifically, the output of STDP

[2] A detailed list of TypeRules is available at: `http://www.csl.sri.com/~shalini/arsenal/` .

Status_attribute-3	: Status_attribute | entity | unique | of=Upper_Desired_Temperature-9
Lower_Desired_Temperature-6	: Lower_Desired_Temperature | entity | unique
Upper_Desired_Temperature-9	: Upper_Desired_Temperature | entity | unique | or: [Lower_Desired_Temperature-6]
equals-10	: equal | predicate | arg2=Invalid-11, arg1=Status_attribute-3
Invalid-11	: Invalid | entity
Regulator_Interface_Failure-14	: Regulator_Interface_Failure | entity | unique
be-16	: be | event
set-17	: set | event | to=True-19, object= Regulator_Interface_Failure-14 | impliedBy: [equals-10]
True-19	: True | bool

Figure 7 IR table for REQ1.

is a set of grammatical relations which lacks logical meaning. To this end, we have developed a semantic processor that takes the output from STDP and systematically applies the type rules to the mentions and dependencies to associate meanings to them, where each type rule specifies a mapping from a set of dependencies (grammatical relations between mentions) to a set of predicates with built-in "semantics."

Consider the following sentence:

If the Regulator_Mode equals INIT, the Heat_Control shall be set to Off. (1)

The result of applying STDP to this sentence is a directed graph that shows the dependencies, as shown in Figure 8.

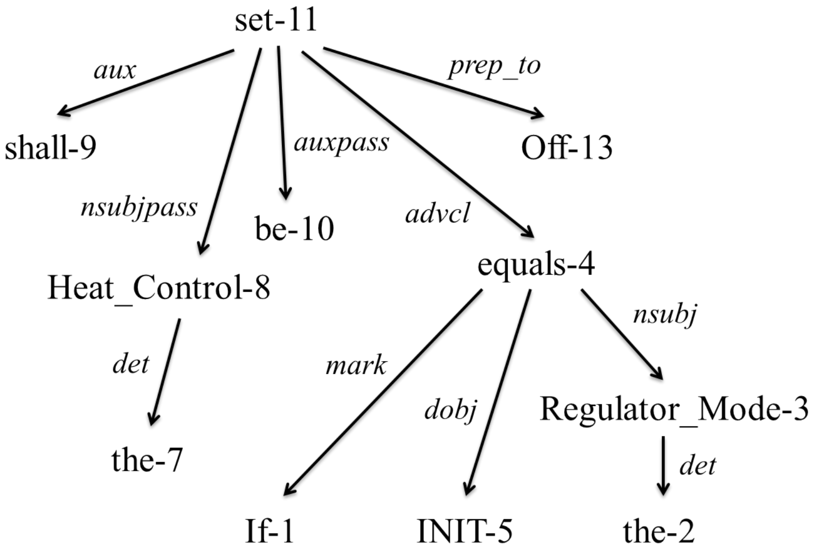


Figure 8 Dependencies generated using STDP.

Applying the set of type rules to the dependency graph generates a predicate graph, as shown in Figure 9. In this predicate graph, similar to the type dependency graph, the edges represent binary predicates (with

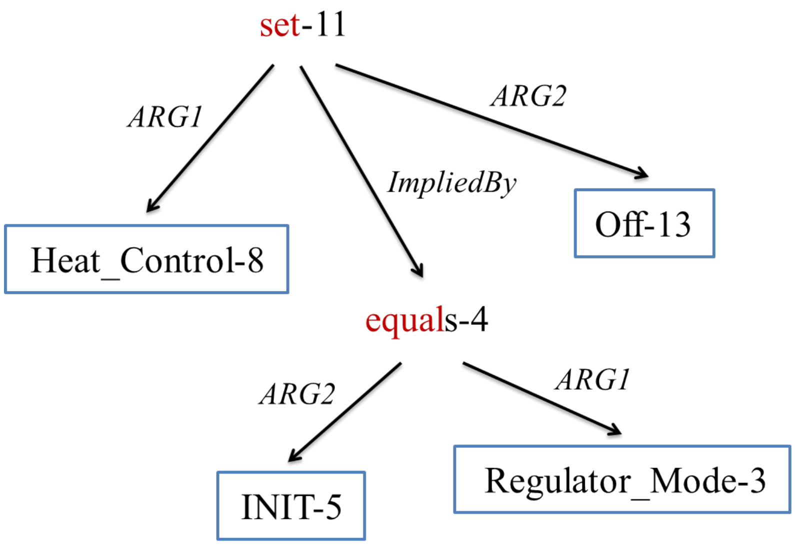


Figure 9 Predicate graph after the application of type rules.

predefined meanings). The unary predicate unique is indicated by boxes with blue borders. Additionally, mentions containing indicative words such as "equal" and "set" (shown in red) are associated with predefined predicates *equal* and *set*. For example, the predicate *set* means that its first argument (ARG1) is a variable being set to a value which is its second argument (ARG2).

5.5 From Predicate Graph to Logical Formulas

This semantic information is further interpreted based on the target language and additional information about the model.[3] For example, to generate an LTL formula from this predicate graph, we can employ the following set of translation rules in Table 2.

Table 2 Formula Translation Rules

Predicate	Expression Translation
$unique(X)$	$tr^u(e(X)) : e(X)$
$set(X) \wedge ARG1(X, Y) \wedge ARG2(X, Z)$	$tr^m(e(X)) : tr^u(e(Y)) = tr^u(e(Z))$
$equal(X) \wedge ARG1(X, Y) \wedge ARG2(X, Z)$	$tr^m(e(X)) : tr^u(e(Y)) = tr^u(e(Z))$
$impliedBy(X, Y)$	$tr^l(e(X)) : tr^m(e(Y)) \rightarrow tr^m(e(X))$

We use $e(X)$ to denote the expression associated with mention X, which we will repeatedly rewrite during the translation process. If mention X is associated with a unique term, i.e. $unique(X)$, its expression is simply the English word in the mention, e.g., e(Heat_Control-8) = Heat_Control. Given a graph representing the predicates over the mentions, we recursively apply the translation rules starting from the root (set-11 in this case). The superscripts m and l indicate the types of the translation rule for simple arithmetic expression and logical expression respectively. When multiple rules are applicable to the same mention, they are applied in the order of tr^l followed by tr^m and then tr^u. The resulting LTL formula after applying the translation rules is shown below.

[3] For example, in a transition system, the variable being set may or may not be a state variable. Hence, additional information may be needed to properly determine the timing of the assignment associated with the predicate *set*.

$$\mathbf{G}\ \big((\text{Regulator\_Mode} = \text{INIT}) \rightarrow (\text{Heat\_Control} = \text{Off})\big) \qquad (2)$$

Every requirement is considered as a *global* requirement except when certain indicative words such as "initialize" are present in the sentence. Hence, we have the **G** operator in front quantifying it for all computations.

6 Formal Analysis

Formal methods have proven effective at providing assurance and finding bugs in a multitude of domains such as electronic designs, software development, and protocol standardization. The *lingua franca* of formal methods is *logic*, which provides an unambiguous semantics for the (formal) language describing a design, and the means to reason with it. However, most people who experience or interact with computers today are "end-users" — they are not necessarily trained in formal methods, and their way of describing their usage to others is through natural language. In many cases, even domain experts, such as circuit designers, resort to natural language as the main medium for communicating their model of a design to consumers of the model (e.g., other designers, implementers and verification engineers), as evidenced by the large proportion of design documents still written in natural language today. Hence, the ability to do formal analysis through a NL interface can bring greater accessibility to the engineering discipline at the requirements stage, by liberating end-users from the burden of learning formal logic. In addition, ARSENAL helps even formal method experts with the ability to create a first formal model quickly and automatically from NL descriptions. We next discuss the different parts of the FM stage (shown in Figure 10).

6.1 Formula Generation

There are multiple *output adapters* in ARSENAL, which convert the IR table (with the semantic metadata annotations) to different output forms. The current ARSENAL implementation includes FOL and LTL adapters, which convert the IR table to FOL and LTL formulas respectively. In this FTR, we discuss the SAL model adapter, which converts the IR table to a SAL model. The SAL model represents a transition system whose semantics is given by a Kripke structure, which is a kind of nondeterministic automaton widely used in model checking [CGP99].

ARSENAL uses translation rules to generate the SAL formulas from the IR table. The translation rules of the SAL adapter are shown in Figures 11 and 12.

The translation from IR to a formula is given by the function `Tf` which is defined inductively starting from the root entry, e.g., `Tf(e(set-17))`. Note that `e` is a function that expands a mention to the IR table entry for that mention. The formula rules in `Tf` invoke translation rules for terms, `Ttl` and `Ttr`, for terms on the left-hand side (LHS) and right-hand side (RHS) of the formula respectively. The translation rules for terms are shown in Figure 11. The main difference between the LHS and RHS term rules is the NMOD rule — this rule generates variables on the LHS but constants on the RHS.[4]

Once the output formula is created from the IR table, we check if the formula includes all the mentions and relations in the IR table. ARSENAL first creates a graph from the IR table, in which each node is a mention entry in the IR table and each (directed) edge indicates if a mention is related to another via relations. It then runs Depth First Search (DFS) on this graph, starting at the root node of the IR table, "coloring" each node as visited as soon as DFS visits that node. When DFS terminates, it checks to see which nodes have not been colored. These nodes are disconnected from the root node and will not be processed by the translation algorithm (and hence not be part of the output formula). ARSENAL shows the uncolored nodes (i.e., uncovered mentions) to the end-user. This approach is very useful for debugging, since it helps to keep track of provenance and coverage of IR mentions in the output formula.

[4] The output trace of applying the translation rules on the IR table is shown in `http://www.csl.sri.com/~shalini/arsenal/`.

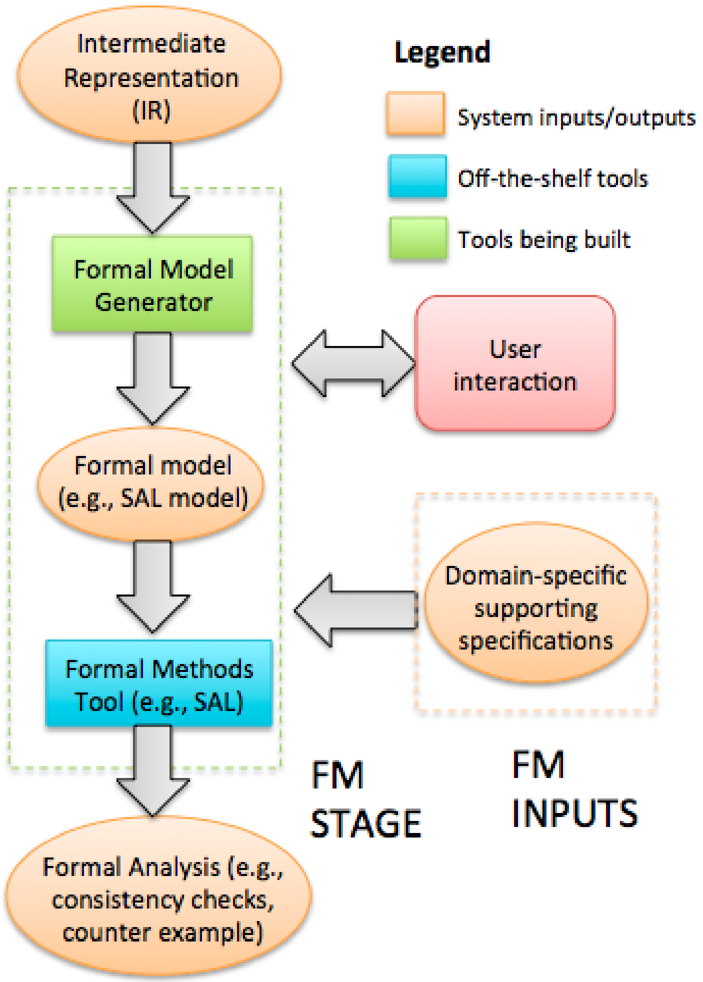

Intermediate Representation (IR)
Legend
System inputs/outputs
Off-the-shelf tools
Tools being built
Formal Model Generator
User interaction
Formal model (e.g., SAL model)
Domain-specific supporting specifications
Formal Methods Tool (e.g., SAL)
FM STAGE
FM INPUTS
Formal Analysis (e.g., consistency checks, counter example)

Figure 10 FM Stage of ARSENAL pipeline.

Rule	IR Entry	Translated Terms (Ttl/Ttr)
VALUE	value | of=X	X (a variable)
DOT	X | entity | of=Y	LHS: Ttl(e(Y)).X RHS: Ttr(e(Y)).X
NUM	X | num	X (a number)
BOOL	X | boolean	X (a boolean)
ARITH	X | arithmetic	X (an arithmetic expression)
NMOD	X | entity | $[M_1, \ldots, M_n]$	LHS: $M_1\_\ldots\_M_n\_X$ (a variable) RHS: $M_1\_\ldots\_M_n\_X$ (a constant)

Figure 11 Translation rules for terms.

Rule	IR Entry	Translated Formula (Tf)
NEG	X | neg	$\neg(\mathrm{Tf}(X))$
G	X | Talways	$\mathbf{G}\ (\mathrm{Tf}(X))$
F	X | Teventually	$\mathbf{F}\ (\mathrm{Tf}(X))$
NEV	X | Tnever	$\neg(\mathbf{F}\ (\mathrm{Tf}(X)))$
UNT	X | Tuntil=Y	$\mathrm{Tf}(X)\ \mathbf{U}\ \mathrm{Tf}(\mathrm{e}(Y))$
AND	X | and:[$Y_1, \ldots, Y_n$]	$\mathrm{Tf}(X) \bigwedge_{i=1}^{n} \mathrm{Tf}(\mathrm{e}(Y_i))$
OR	X | or:[$Y_1, \ldots, Y_n$]	$\mathrm{Tf}(X) \bigvee_{i=1}^{n} \mathrm{Tf}(\mathrm{e}(Y_i))$
IMP	X | impliedBy: Y	$\mathrm{Tf}(\mathrm{e}(Y)) \Rightarrow \mathrm{Tf}(X)$
EQ	equals | ... | args=$\langle X, Y\rangle$	$\mathrm{Ttl}(\mathrm{e}(X)) = \mathrm{Ttr}(\mathrm{e}(Y))$
EXC	exceed | ... | args=$\langle X, Y\rangle$	$\mathrm{Ttl}(\mathrm{e}(X)) > \mathrm{Ttr}(\mathrm{e}(Y))$
DOM	dominate | args=$\langle X, Y\rangle$	$\mathrm{Ttl}(\mathrm{e}(X)) \geq \mathrm{Ttr}(\mathrm{e}(Y))$
ATOM	X (only)	A boolean formula generated from X
SET	set | ... | obj=X, to=Y	$\mathrm{Ttl}(\mathrm{e}(X)) = \mathrm{Ttr}(\mathrm{e}(Y))$
SET1	set | ... | obj=X	$\mathrm{Ttl}(\mathrm{e}(X)) = 1$
CLR	clear | ... | obj=X	$\mathrm{Ttl}(\mathrm{e}(X)) = 0$
INIT	initialize | ... | obj=X, to=Y	$\mathrm{Ttl}(\mathrm{e}(X)) = \mathrm{Ttr}(\mathrm{e}(Y))$
SEND	send | obj=X	out_channel = $\mathrm{Ttr}(\mathrm{e}(X))$
TRAN	transmit | obj=X	out_channel = $\mathrm{Ttr}(\mathrm{e}(X))$
REC	receive | obj=X	in_channel = $\mathrm{Ttr}(\mathrm{e}(X))$
IN	be | ... | agent=X, in=Y	$\mathrm{Ttl}(\mathrm{e}(X)) = \mathrm{Ttr}(\mathrm{e}(Y))$

Figure 12 Translation rules for formulas.

6.2 Formal Analysis of Generated Formulas

We do the formal analysis of the generated formulas using Büchi Automata. A Büchi automaton extends a finite automaton to infinite inputs. It is an ω-automaton $A = (Q, \Sigma, \delta, q0, F)$ that consists of the following components:

- Q is a finite set, whose elements are called the states of A.
- Σ is a finite set called the alphabet of A.
- $\delta : Q \times \Sigma \rightarrow Q$ is a function, called the transition function of A.
- $q0$ is an element of Q, called the initial state.
- $F \subseteq Q$ is the acceptance condition. A accepts exactly those runs in which at least one of the infinitely often occurring states is in F.

Büchi automata is used in model checking, as an automata-theoretic version of a formula in LTL. If the Buchi Automata generated from a set of formulas is degenerate, the formulas are inconsistent. Given a set of LTL formulas generated by the Formula Generator, we first check if the Büchi Automata created from corresponding propositionalized formulas is degenerate — if so, we report the inconsistency in the requirements to the user. If not, we proceed to creating the SAL model, as described next.

6.3 SAL Model Generation

SAL can be used to prove theorems, encoding properties about the requirements, using bounded model-checking. If SAL finds a counter-example, we know the property does not hold. If SAL does not find a counter-example at a known depth of model-checking, we try to see if the LTL formulas are realizable.

We continue to use REQ1 to illustrate how ARSENAL produces a SAL model from the generated formulas in the previous step. At its core, SAL is a language for specifying transition systems in a compositional way. A transition system is composed (synchronously or asynchronously) of modules, where each module consists of a *state type*, an *invariant definition* on this state type, an *initialization condition* on the state type, and a *binary transition relation* on the state type. The state type is defined by four pairwise disjoint types of variables: *input*, *output*, *local* and *global* — input and global variables are observed variables of a module, while output, global and local variables are controlled variables of a module. Note that the SAL model-checkers use LTL, a modal temporal logic with modalities referring to time, as their underlying assertion language. This is an appropriate language for formally expressing requirements, since many requirements have temporal operators (e.g., eventually, always).

In order to unambiguously define a transition system, we need to additionally distinguish *controlled* variables that are *state* variables from *wires* (variables that do not directly define the state space). We need to know the type of any variable. Since we consider only self-contained modules in SAL, a variable can then belong to one of the following five categories: *input, state only, state and output, output only*, and *wire*. By differentiating *state* variables from *wires*, we can unambiguously map them to the corresponding section in SAL, namely DEFINITION or TRANSITION. We use the former to describe constraints over wires, and the latter to describe evolutions of state variables. For example, if the `Regulator_Interface_Failure` variable in REQ1 was a *state* variable, then the SAL model generator would have produced the following transition instead.

```
TRANSITION
  Upper_Desired_Temperature.Status_attribute = Invalid
  OR Lower_Desired_Temperature.Status_attribute =
  Invalid --> Regulator_Interface_Failure' = TRUE
```

The SAL model would hence be different, even though generated from the same sentence. Currently, ARSENAL requires the user to provide this additional semantic information only after the NLP stage, thus keeping it separate from the model-independent part of the pipeline.

Table 3 Rules for Gathering Type Evidence

Expression	Inference
$X \bowtie Y$, $\bowtie \in \{<, >, \leq, \geq\}$	X and Y are numbers
$X =$ a number	X is a number
$X =$ a named value C	X has enum type containing C
$X = Y$	X and Y have same type

During the model generation stage, ARSENAL gathers type evidences for each variable across all sentences and performs type inference by organizing them into equivalence classes. Further, in case of a type conflict, a warning is produced to indicate inconsistency in the NL sentences, thus helping the user to refine their requirements documentation at an early stage. Table 3 summarizes the rules ARSENAL currently implements for gathering type evidence.

6.4 Verification and Synthesis with LTL

In addition to specifying a transition system, sentences in NL may describe high-level requirements that the system must satisfy. Often times, this requirement can be precisely captured using temporal logic. In this FTR, we use LTL [MP92], whose semantics can be interpreted over Kripke structures. We consider two problems that leverage LTL to reason about potential inconsistencies in a NL documentation, namely verification and synthesis. For verification, we use model checking to analyze whether a SAL model satisfies its LTL specification. In general, the application of model-checking tools involves a nontrivial step of creating a mathematical model of the system and translating the desired properties into a formal specification. ARSENAL automates this process with minimal user guidance. Given a model M as a (labeled) transition system and a specification ψ in LTL, both produced in the NLP stage of ARSENAL, we check if $M \models \psi$. When the model does not satisfy the specification, a negative answer (often in the form of a counterexample) is presented to the user as a certificate of how the system fails the specification. In this FTR, we use SAL's bounded model checking [CBRZ01] capability as the main workhorse for finding such inconsistencies.

ARSENAL also provides the flexibility to generate only specifications from the natural language requirements. Consistency means the specification is *satisfiable*, that is, whether a model exists for the specification. This problem is known as LTL satisfiability checking and it can be reduced to model checking [RV07]. Given an LTL formula ψ, it is satisfiable precisely when a *universal* model M[5] does not satisfy $\neg\psi$. A counterexample that points to the inconsistency is produced when ψ is not satisfiable.

Given an LTL specification, it may also be possible to directly *synthesize* an implementation that satisfies the specification. *Realizability*, the decision problem of determining whether such an implementation exists, can be used to further inspect the requirements document for inconsistencies. If the specification is realizable, then a Moore machine can be extracted as an implementation that satisfies the specification. Thus, the benefit of LTL synthesis is a correct-by-construction process that can automatically generate an implementation from its specification. In general, LTL synthesis has high computational complexity [Kup12]. However, it has been shown that a subclass of LTL, known as Generalized Reactivity (1) [GR(1)], is more amenable to synthesis [PP06] and is also expressive enough for specifying complex industrial designs [BGJ$^+$07].

Formal specification can precisely capture the desired properties of a design. However, it is common for formal specifications to be incomplete. Assumptions or constraints about the environment are particularly hard to capture. In this section, we describe a technique to generate *candidate environment assumptions* as suggestive solutions to make an unrealizable specification realizable. This is motivated by the fact that, in many scenarios, simply producing an *unrealizable* answer is not very useful for an user. Playing a two-player game according to the counterstrategy can be useful [BCG$^+$10], but it requires considerable effort and time, not to mention the expertise in formal method that an user is assumed to have. To overcome this problem, we propose finding potentially missing assumptions about the environment, and then recommending them to the user as NL sentences in an interactive way. Throughout the process, the user remains oblivious to the underlying formal analysis performed, and can just reason with the NL feedback directly.

Given a LTL specification ψ that is satisfiable but not realizable, the assumption mining problem is to find ψ_a such that $\psi_a \rightarrow \psi$ is realizable. Our algorithm for computing ψ_a follows the counterstrategy-guided approach in [BCG$^+$10], which has shown to be able to generate useful and intuitive environment assumptions for digital circuits and robotic controllers. The algorithm is based on [LDS11], and is summarized below.

Counterstrategy-guided synthesis of environment assumptions. Given the unrealizable specification ψ, the method first computes a counterstrategy. The counterstrategy summarizes the next moves of the environment in response to the current output of the system, which will force a violation of the specification. The method then uses a template-based mining approach to find a specification ϕ that is satisfied by the counterstrategy. $\neg\phi$ is added as a new conjunct to ψ_a and $\psi_a \wedge \psi_e \rightarrow \psi_s$ is checked for realizability again. By asserting the negation of ϕ as an assumption to the original specification, the method effectively eliminates the moves by the environment that adhere to the counterstrategy. The process iterates until the resulting specification becomes realizable. At any step of the iteration, the user is asked to verify the correctness of the mined assumption. We present them as NL sentences, which we generate by mapping the boolean and temporal operators to English connectives.

In the rest of the section, we give details of the realizability analysis of LTL specifications, and demonstrate some of the benefits of these techniques when they are incorporated into ARSENAL and applied to different corpora. We envision ARSENAL to be applicable in a multitude of formal analysis settings. Figure 13 illustrates one such scenario, where ARSENAL generates LTL specifications from natural language.

LTL Synthesis Given a LTL specification ϕ, LTL synthesis is the process of automatically finding an implementation that satisfies ϕ. *Realizability* is the question of asking if such implementation exists. In our tool flow, if the specification is not realizable, there is a suite of techniques we can employ to explain the problem of unrealizability, thereby helping the designers to debug their design at the requirement stage. Synthesis for full LTL can be prohibitive expensive (2EXPTIME-Complete). However, there exists efficient algorithms for handling a subclass of LTL, known as Generalized Reactivity (1). We describe the syntax of GR(1) below. In general, the algorithm works on specifications (assumptions and requirements) that are given as deterministic Büchi Word Automata as well. Given input/output partitions of the Boolean variables into $I \cup O$, GR(1) specification takes the form $\phi_e \rightarrow \phi_s$, where each ϕ_i is a conjunction of the following:

[5] Universal means M contains all possible traces over the set of atomic propositions.

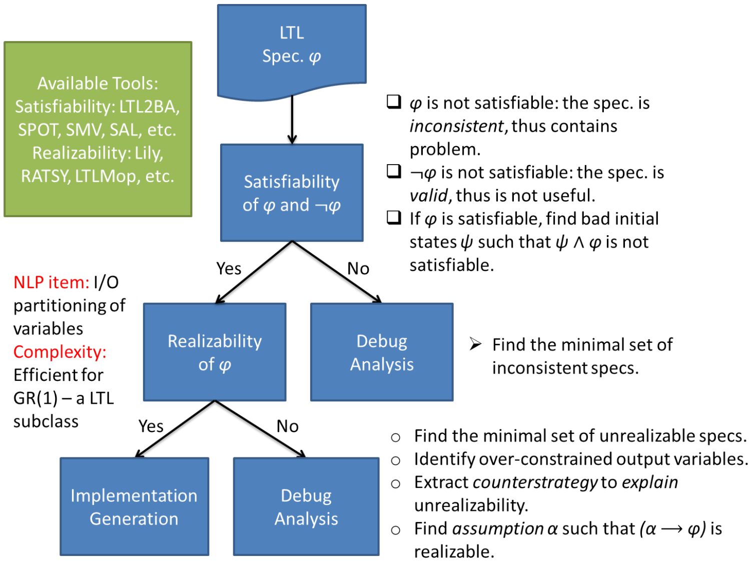

Available Tools:
Satisfiability: LTL2BA, SPOT, SMV, SAL, etc.
Realizability: Lily, RATSY, LTLMop, etc.
LTL Spec. φ
Satisfiability of φ and ¬φ
❑ φ is not satisfiable: the spec. is *inconsistent*, thus contains problem.
❑ ¬φ is not satisfiable: the spec. is *valid*, thus is not useful.
❑ If φ is satisfiable, find bad initial states ψ such that ψ ∧ φ is not satisfiable.
Yes
No
NLP item: I/O partitioning of variables
Complexity: Efficient for GR(1) – a LTL subclass
Realizability of φ
Debug Analysis
➢ Find the minimal set of inconsistent specs.
Yes
No
Implementation Generation
Debug Analysis
○ Find the minimal set of unrealizable specs.
○ Identify over-constrained output variables.
○ Extract *counterstrategy* to *explain* unrealizability.
○ Find *assumption* α such that (α ⟶ φ) is realizable.

Figure 13 LTL satisfiability and synthesis using ARSENAL output.

– Initial states: α_i, which is a Boolean formula that characterizes the initial states of the environment/system.
– Transitions: β_i, which is a formula of the form $\bigwedge_j \mathbf{G}\, f_j$ where each f_j is a Boolean combination of variables from $I \cup O$ and expressions of the form $\mathbf{X}\, v$ where $v \in I$ if $i = e$, and $v \in I \cup O$ otherwise.
– Fairness: γ_i, which is a formula of the form $\bigwedge_j \mathbf{G}\,\mathbf{F}\, f_j$, where each f_j is a Boolean formula.

From the NLP perspective, we need additional knowledge on how the variables are partitioned as inputs or outputs. Currently, this part is done by manual annotations of the variables. We evaluate the usefulness of LTL synthesis on actual English requirements, extracted from the Appendix A of a FAA requirement document. We detail our analysis below.

6.5 Realizability Analysis: Case Study on FAA-Isolette

In this section we describe a study on requirements from an Isolette design, focusing on realizability. An Isolette is an incubator for infants that provides controlled temperature, humidity and oxygen. Our example is an Isolette Thermostat that regulates the air temperature inside an Isolette such that it is maintained within a desired range. The Thermostat is composed of two interacting modules – the "Regulate Temperature" module and the "Monitor Temperature" module. We focus on the "Regulate Temperature" module, which receives input from the "Operator Interface" and the "Monitor Temperature" module, and produces output to the "Heat Source". The requirements are taken from Appendix A of the "Requirement Engineering Management Handbook" released by the Federal Aviation Administration, which was intended to serve as an example of the "best practices" advocated in this handbook [LM09a]. The English sentences describing the requirements, as well as their sources in the document, are tabularized in Table 4.

Additionally, we require the user to provide additional type information for each variable. For this FAA-Isolette example, we assume the following type information is given.

– *Input:*
  - Upper_Desired_Temperature_Status
  - Lower_Desired_Temperature_Status
  - Regulator_Init_Timeout
  - Current_Temperature

Table 4 Isolette requirements in English

	Requirement in English	Source
1	If the Regulator Mode equals INIT, the Output Regulator Status shall be set to Init.	REQ-MRI-1
2	If the Regulator Mode equals NORMAL, the Output Regulator Status shall be set to On.	REQ-MRI-2
3	If the Regulator Mode equals FAILED, the Output Regulator Status shall be set to Failed.	REQ-MRI-3
4	If the Status attribute of the Lower Desired Temperature or the Upper Desired Temperature equals Invalid, the Regulator Interface Failure shall be set to True.	REQ-MRI-6
5	If the Status attribute of the Lower Desired Temperature and the Upper Desired Temperature equals Valid, the Regulator Interface Failure shall be set to False.	REQ-MRI-7
6	If the Regulator Mode equals INIT, the Heat Control shall be set to Off.	REQ-MHS-1
7	If the Regulator Mode equals NORMAL, and the Current Temperature is less than the Lower Desired Temperature, the Heat Control shall be set to Control On.	REQ-MHS-2
8	If the Regulator Mode equals NORMAL, and the Current Temperature is greater than the Upper Desired Temperature, the Heat Control shall be set to Control Off.	REQ-MHS-3
9	If the Regulator Mode equals FAILED, the Heat Control shall be set to Control Off.	REQ-MHS-5
10	If the Regulator Interface Failure is set to False, and the Regulator Internal Failure is set to False, and the Status attribute of the Current Temperature is set to Valid, the Regulator Status shall be set to True.	Table A-10
11	If the Regulator Interface Failure is set to True or the Regulator Internal Failure is set to True or the Status attribute of the Current Temperature is not set to Valid, the Regulator Status shall be set to False.	Table A-10
12	The Regulator Mode shall be initialized to INIT.	Req MRM 1
13	If the Regulator Mode equals INIT and the Regulator Status equals True, the Regulator Mode shall be set to NORMAL.	Req MRM 2
14	If the Regulator Mode is set to NORMAL and the Regulator Status is set to False, the Regulator Mode shall be set to FAILED.	Req MRM 3
15	If the Regulator Mode is set to INIT and the Regulator Init Timeout is set to True, the Regulator Mode shall be set to FAILED.	Req MRM 4

  - Lower_Desired_Temperature
  - Upper_Desired_Temperature
  - Regulator_Internal_Failure
  - Current_Temperature_Status
  - Regulator_Interface_Failure
  - Regulator_Status
- *State and output:* Regulator_Mode
- *Pure output:* Output_Regulator_Status, Heat_Control

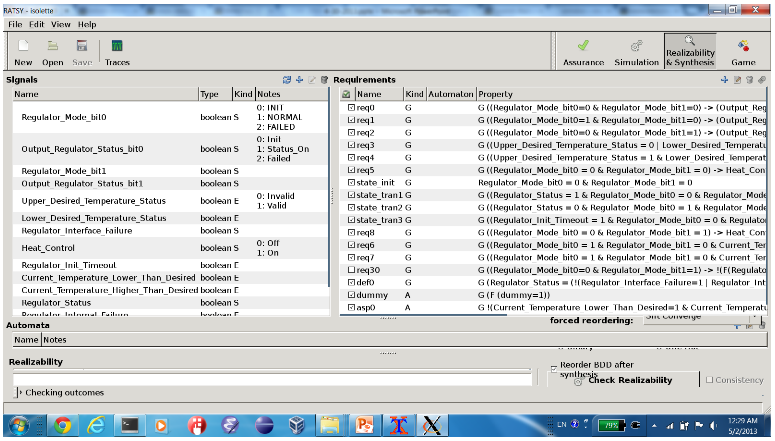
Using this information, we are now ready to generate LTL [GR(1)] formulas. The generated LTL formulas corresponding to the English sentences are listed below. In addition, we automatically determine which terms correspond to variables and which terms correspond to values (based on predicate such as *set*) and gather domain information (variables assumed to have enumerative type) across the sentences for all the variables.

1. $\mathbf{G}\ \big((\text{Regulator\_Mode} = \text{INIT}) \rightarrow (\text{Output\_Regulator\_Status} = \text{Init})\big)$
2. $\mathbf{G}\ \big((\text{Regulator\_Mode} = \text{NORMAL}) \rightarrow (\text{Output\_Regulator\_Status} = \text{On})\big)$
3. $\mathbf{G}\ \big((\text{Regulator\_Mode} = \text{FAILED}) \rightarrow (\text{Output\_Regulator\_Status} = \text{Failed})\big)$
4. $\mathbf{G}\ \big((\text{Upper\_Desired\_Temperature\_Status} = \text{Invalid} \vee \text{Lower\_Desired\_Temperature\_Status} = \text{Invalid}) \rightarrow (\text{Regulator\_Interface\_Failure} = \texttt{true}\ )\big)$
5. $\mathbf{G}\ \big((\text{Upper\_Desired\_Temperature\_Status} = \text{Valid} \wedge \text{Lower\_Desired\_Temperature\_Status} = \text{Valid}) \rightarrow (\text{Regulator\_Interface\_Failure} = \texttt{false}\ )\big)$
6. $\mathbf{G}\ \big((\text{Regulator\_Mode} = \text{INIT}) \rightarrow (\text{Heat\_Control} = \text{Control\_Off})\big)$
7. $\mathbf{G}\ \big((\text{Regulator\_Mode} = \text{NORMAL} \wedge \text{Current\_Temperature} < \text{Lower\_Desired\_Temperature} = \texttt{true}\ ) \rightarrow (\text{Heat\_Control} = \text{Control\_On})\big)$
8. $\mathbf{G}\ \big((\text{Regulator\_Mode} = \text{NORMAL} \wedge \text{Current\_Temperature} > \text{Upper\_Desired\_Temperature} = \texttt{true}\ ) \rightarrow (\text{Heat\_Control} = \text{Control\_Off})\big)$
9. $\mathbf{G}\ \big((\text{Regulator\_Mode} = \text{FAILED}) \rightarrow (\text{Heat\_Control} = \text{Control\_Off})\big)$
10. $\mathbf{G}\ \big((\text{Regulator\_Interface\_Failure} = \texttt{false} \wedge \text{Regulator\_Internal\_Failure} = \texttt{false} \wedge \text{Current\_Temperature\_Status} = \text{Valid}) \rightarrow (\text{Regulator\_Status} = \texttt{true}\ )\big)$
11. $\mathbf{G}\ \big((\text{Regulator\_Interface\_Failure} = \texttt{true} \vee \text{Regulator\_Internal\_Failure} = \texttt{true} \vee \neg(\text{Current\_Temperature\_Status} = \text{Valid})) \rightarrow (\text{Regulator\_Status} = \texttt{false}\ )\big)$
12. $\text{Regulator\_Mode} = \text{INIT}$
13. $\mathbf{G}\ \big((\text{Regulator\_Mode} = \text{INIT} \wedge \text{Regulator\_Status} = \texttt{true}\ ) \rightarrow \mathbf{X}\ (\text{Regulator\_Mode} = \text{NORMAL})\big)$
14. $\mathbf{G}\ \big((\text{Regulator\_Mode} = \text{NORMAL} \wedge \text{Regulator\_Status} = \texttt{false}\ ) \rightarrow \mathbf{X}\ (\text{Regulator\_Mode} = \text{FAILED})\big)$
15. $\mathbf{G}\ \big((\text{Regulator\_Mode} = \text{INIT} \wedge \text{Regulator\_Init\_Timeout} = \texttt{true}\ ) \rightarrow \mathbf{X}\ (\text{Regulator\_Mode} = \text{FAILED})\big)$

We are interested in the question of whether there exists a Finite State Machine (FSM) implementation that can satisfy all the specifications/theorems for all inputs. Specifically, we use RATSY [BCG^{+}10] to check realizability of the specifications. Since RATSY only supports Boolean variables, we propositionalize the specifications. For example, Regulator_Mode has an enumerated type with values $\{\text{INIT}, \text{NORMAL}, \text{FAILED}\}$. In RATSY, we use two bits Regulator_Mode_bit0 and Regulator_Mode_bit1 to encode this variable. In general, LTL synthesis also requires an input/output partition on the set of signals. In this case, we make all wires and controls (explicit) output as well. Figure 14 shows a snapshot of the specifications written in RATSY.

Given that ARSENAL was able to produce a SAL model from the same set of specifications earlier, one would expect that there should exist a FSM that can implement these specifications. However, RATSY reports that the specifications are not realizable, as shown in Figure 15.

Recall that realizability concerns with the question of whether the specifications can be implemented. The reason that these specifications are not realizable while ARSENAL could still generate a SAL model from them was because the SAL model was nondeterministic. Specifically, when Regulator_Status is TRUE



Signals

Name	Type	Kind	Notes
Regulator_Mode_bit0	boolean	S	0: INIT 1: NORMAL 2: FAILED
Output_Regulator_Status_bit0	boolean	S	0: Init 1: Status_On 2: Failed
Regulator_Mode_bit1	boolean	S	
Output_Regulator_Status_bit1	boolean	S	
Upper_Desired_Temperature_Status	boolean	E	0: Invalid 1: Valid
Lower_Desired_Temperature_Status	boolean	E	
Regulator_Interface_Failure	boolean	S	
Heat_Control	boolean	S	0: Off 1: On
Regulator_Init_Timeout	boolean	E	
Current_Temperature_Lower_Than_Desired	boolean	E	
Current_Temperature_Higher_Than_Desired	boolean	E	
Regulator_Status	boolean	S	
Regulator_Internal_Failure	boolean	E	

Requirements

☑	Name	Kind	Automaton	Property
☑	req0	G		G ((Regulator_Mode_bit0=0 & Regulator_Mode_bit1=0) -> (Output_Reg
☑	req1	G		G ((Regulator_Mode_bit0=1 & Regulator_Mode_bit1=0) -> (Output_Reg
☑	req2	G		G ((Regulator_Mode_bit0=0 & Regulator_Mode_bit1=1) -> (Output_Reg
☑	req3	G		G ((Upper_Desired_Temperature_Status = 0 | Lower_Desired_Temperatu
☑	req4	G		G ((Upper_Desired_Temperature_Status = 1 & Lower_Desired_Temperat
☑	req5	G		G ((Regulator_Mode_bit0 = 0 & Regulator_Mode_bit1 = 0) -> Heat_Con
☑	state_init	G		Regulator_Mode_bit0 = 0 & Regulator_Mode_bit1 = 0
☑	state_tran1	G		G ((Regulator_Status = 1 & Regulator_Mode_bit0 = 0 & Regulator_Mode
☑	state_tran2	G		G ((Regulator_Status = 0 & Regulator_Mode_bit0 = 1 & Regulator_Mode
☑	state_tran3	G		G ((Regulator_Init_Timeout = 1 & Regulator_Mode_bit0 = 0 & Regulator
☑	req8	G		G ((Regulator_Mode_bit0 = 0 & Regulator_Mode_bit1 = 1) -> Heat_Con
☑	req6	G		G ((Regulator_Mode_bit0 = 1 & Regulator_Mode_bit1 = 0 & Current_Ter
☑	req7	G		G ((Regulator_Mode_bit0 = 1 & Regulator_Mode_bit1 = 0 & Current_Ter
□	req30	G		G ((Regulator_Mode_bit0=0 & Regulator_Mode_bit1=1) -> !(F(Regulato
☑	def0	G		G (Regulator_Status = (!(Regulator_Interface_Failure=1 | Regulator_Int
☑	dummy	A		G (F (dummy=1))
☑	asp0	A		G !(Current_Temperature_Lower_Than_Desired=1 & Current_Temperatu

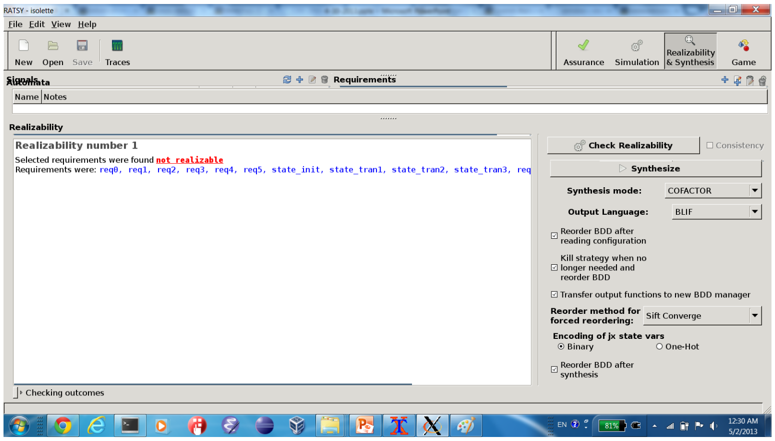
Automata

Name	Notes

Realizability

☑ Reorder BDD after synthesis

Figure 14 Specifications in RATSY.



Realizability number 1

Selected requirements were found not realizable
Requirements were: req0, req1, req2, req3, req4, req5, state_init, state_tran1, state_tran2, state_tran3, req

Synthesis mode: COFACTOR

Output Language: BLIF

☑ Reorder BDD after reading configuration

☑ Kill strategy when no longer needed and reorder BDD

☑ Transfer output functions to new BDD manager

Reorder method for forced reordering: Sift Converge

Encoding of jx state vars
● Binary ○ One-Hot

☑ Reorder BDD after synthesis

Figure 15 Specifications are not realizable as reported by RATSY.

and Regulator_Init_Timeout is TRUE at the same time, the requirements do not specify which mode to transition to, as illustrated in Figure 16.

RATSY also provides a feature for debugging unrealizable specifications, by having the user playing against the *counterstrategy* (winning strategy for the environment), as shown in Figure 17.

Counterstrategy is presented as a game between the user and the tool. The tool sets the inputs and the user sets the outputs. The game is played by the tool first setting inputs according to the counterstrategy, and then the user trying to satisfy all the requirements by setting the outputs (which is impossible). In this process, the user can understand the cause of unrealizability.

A common cause of unrealizability is an under-constrained environment. We have incorporated the counterstrategy-guided assumption mining approach developed by Li et al. in [LDS11]. This technique uses a template generalization approach to find candidate assumptions that can rule out the counterstrategy of an unrealizable specification, thereby guiding the specification towards realizability. In this case, the following assumption is mined and it makes the conflict go away such that the specification is realizable.

$$\phi := \mathbf{G}\ \neg(\text{Regulator\_Status} = \texttt{true}\ \wedge \text{Regulator\_Init\_Timeout} = \texttt{true}\ ) \tag{3}$$

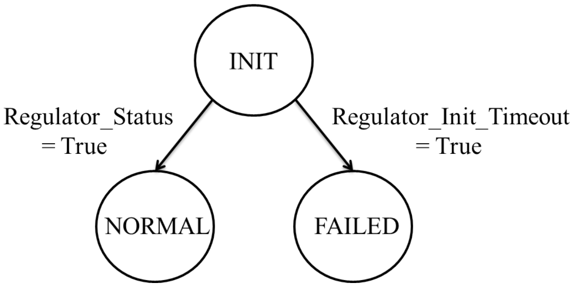
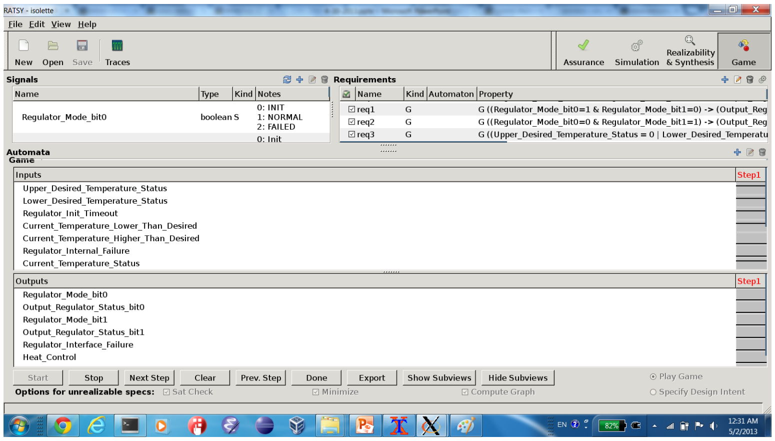


Figure 16 Transition of Regulator_Mode based on Requirement 13 and 15.



Signals

Name	Type	Kind	Notes
Regulator_Mode_bit0	boolean	S	0: INIT 1: NORMAL 2: FAILED
			0: Init

Requirements

☑	Name	Kind	Automaton	Property
☑	req1	G		G ((Regulator_Mode_bit0=1 & Regulator_Mode_bit1=0) -> (Output_Reg
☑	req2	G		G ((Regulator_Mode_bit0=0 & Regulator_Mode_bit1=1) -> (Output_Reg
☑	req3	G		G ((Upper_Desired_Temperature_Status = 0 | Lower_Desired_Temperatu

Automata

Game

Inputs	Step1
Upper_Desired_Temperature_Status	
Lower_Desired_Temperature_Status	
Regulator_Init_Timeout	
Current_Temperature_Lower_Than_Desired	
Current_Temperature_Higher_Than_Desired	
Regulator_Internal_Failure	
Current_Temperature_Status	

Outputs	Step1
Regulator_Mode_bit0	
Output_Regulator_Status_bit0	
Regulator_Mode_bit1	
Output_Regulator_Status_bit1	
Regulator_Interface_Failure	
Heat_Control	

Options for unrealizable specs: ☑ Sat Check ☑ Minimize ☑ Compute Graph

● Play Game
○ Specify Design Intent

Figure 17 Debugging unrealizable specifications in RATSY.

This assumption can be presented to the user as a recommendation. If the user rejects this assumption, another assumption (or set of assumptions) will be produced until either the specification is realizable or all the recommendations are rejected by the user.

Since now the specification is realizable, RATSY can synthesize a design that implements the specification. The corresponding Verilog file can be found at `http://www.csl.sri.com/users/shalini/arsenal/faa-isolette.v`.

In addition to the kind of analysis presented above, we are considering the following directions for future work. In general, there is a need to challenge positive answers in verification and synthesis as well. We list them below and plan to explore them more in future work.

– All verification properties passed: still need to check *vacuity*, e.g. a property is vacuously true, and *coverage*, e.g. mutation-based coverage for model checking.
– The properties are realizable: how do I know the generated implementation is correct (complete)? Maybe we can run a separate test suite to verify the synthesized design?
– Incremental LTL synthesis is still an open problem: if properties are modified/added, one has to redo synthesis.

7 Results and Discussion

In this section, we present results on analyzing the FAA-Isolette corpus [LM09b] and a portion of the TTEthernet requirements document [SD10] to demonstrate ARSENAL's ability in handling complex NL sentences and different corpora. To better understand the degree of automation and robustness ARSENAL can achieve, we separately evaluate different stages of the ARSENAL pipeline.

7.1 NLP Stage: Evaluation

Degree of Automation Metric In this section, we report automation results of ARSENAL on both the FAA-Isolette (FAA) and the TTEthernet (TTE) corpora. Specifically, we evaluate the accuracy of ARSENAL's NLP pipeline on translating each NL sentence into the corresponding logical formula automatically, without any manual correction. This metric measures the degree to which ARSENAL runs in an automated mode.

The results are summarized in Table 5. When evaluating accuracy, the correct outputs were given a score of 1.0, wrong outputs were given a score of 0.0, while partially correct results were given partial credit of 0.5. A translation was deemed partially correct if there was one error and incorrect if there was more than one error in the resulting formula.

Table 5 ARSENAL NLP pipeline accuracy.

Corpus	Total	Correct	Partial	Wrong	Degree of Automation
TTE	36	24	8	4	78%
FAA	42	39	2	1	95%

Note that when ARSENAL fails to give the correct output automatically from the NLP stage while processing requirements, we correct the error manually so that the input to the FM stage is correct. The following sentence is one of the sentences in FAA for which ARSENAL partially captures the logical semantics.

REQ2: *If the Regulator Mode equals NORMAL, the Temp attribute of the Display Temperature shall be set to Temp attribute of the Current Temperature rounded to the nearest integer.*

The logical formula output by ARSENAL is:

```
(Regulator_Mode = NORMAL => Display_Temperature.Temp_attribute =
Current_Temperature.Temp_attribute)
```

The reason ARSENAL only handles the first half of the sentence correctly is that the phrase "rounded to the nearest integer" implies there is a function that can take a real/floating-point number as input and produce its *nearest* integer as output. Currently, ARSENAL does not have support for arbitrary functions — in the future, we plan to incorporate more domain-specific knowledge and have built-in support for frequently occurring functions.

Degree of Perturbation Metric We define an evaluation criteria for measuring the robustness of ARSENAL, i.e., if perturbations/modifications are made to a requirements sentence using certain rewrite rules, whether ARSENAL can still generate the right output formula. For the given dataset (e.g., FAA or TTE), we do perturbations to the requirements in that dataset using a transformational grammar, having operators that transform the text. The transformations in this grammar are based on allowable terminals in SAL, e.g., we can replace "always" by "eventually", "or" by "and", "is" by "is not", etc. By applying these transformation operators to the FAA dataset, we can generate a "perturbed" dataset. This is similar in principle to generating test cases by fuzz testing [GLM08]. Note that transforming "or" to "and" can be a significant perturbation for ARSENAL if a sentence has a combination of "and" and "or" terms, since the transformation can change the nesting structure of clauses in the output formula.

Table 6 shows the results of our experiments on the FAA and TTE datasets. Note that total number of requirements was 42 in FAA and 36 in TTE. Out of the 36 requirements in TTE, the "And → Or" rewrite rule affected 16 requirements. We ran two types of "And → Or" transformations — in the first case, we modified only the first occurrence of "And" in the requirements sentences, while in the second case we modified

Table 6 Results of perturbation test on ARSENAL.

Perturbation Type	TTEthernet domain (TTE)			FAA-Isolette domain (FAA)		
	Total sentences	Perturbed sentences	Accuracy	Total sentences	Perturbed sentences	Accuracy
First (And→Or)	36	16	81%	42	N/A	N/A
All (And→Or)	36	16	87%	42	13	92%
All (Is→Is not)	36	17	100%	42	13	92%
If A then B→B if A	36	N/A	N/A	42	40	65%

all occurrences of "And" in the sentences. When ARSENAL was run on these transformed requirements, thirteen of them gave output formulas that were correct w.r.t. the modified requirements sentence for the "First (And → Or)" rewrite rule, while fourteen of them gave output formulas that were correct for the "All (And → Or)" rewrite rule, giving an accuracy of $13/16 \approx 81\%$ and $14/16 \approx 87\%$ respectively. Similar numbers were calculated for other rules on FAA and TTE. For FAA, only 2 sentences had more than one AND in them — so we did not run the "First (And→Or)" transformation on FAA, since the results for that would have been quite close to the "All (And→Or)" rule.

We subsequently tried more complex rewrite rules, e.g., of the form "If A then B→B if A". For the "If A then B→B if A" rule, ARSENAL's lower accuracy of 65% on the FAA domain was mainly caused by incorrect parse output from STDP on the perturbed sentences. For TTE, none of the 36 sentences had the "If A then B" structure.

NLP Stage Accuracy Our goal here is evaluating the accuracy of the NLP stage of ARSENAL. We take the approach of estimating how many sub-formulas are inserted, deleted or modified by ARSENAL in the NLP stage, while generating the output formula for a requirements sentence.

To this effect, we designed a novel metric. We first considered a corpus of requirements sentences that have been annotated with the expected output formula, which we call the *ground-truth corpus*. We used the following algorithm to compute this metric:

1. Given each requirements sentence in this ground-truth corpus, generate all sub-formulas G of the ground-truth formula and all sub-formulas A of the formula generated by ARSENAL.
2. For each sub-formula in A, find the best-matching sub-formulas in G using a new algorithm: *Max-weighted matching in bipartite graphs*. The match score between formulas is calculated using a variant of string edit distance that we designed, suitably modified for formulas, called *Typed Levenshtein distance.*
3. Calculate precision/recall/F-measure using the similarity scores between matched pairs of sub-formulas.

Typed Levenshtein Distance This is a modified version of the Levenshtein string edit distance, where a higher-level token-based edit distance calls an underlying character-based edit distance. The computation of that score is outlined below:

```
d[i][j] =
  if seq1[i] == seq2[j]
    then d[i+1][j+1]
  else min(
    IC + d[i+1][j],
    DC + d[i][j+1],
    c(seq1[i], seq2[j]) + d[i+1][j+1])
```

Here, d is the distance, IC is insertion cost, DC is the deletion cost, and seq1 and seq2 are the two sequences. We first turn the formula into a sequence of tokens, with 3 types of tokens:

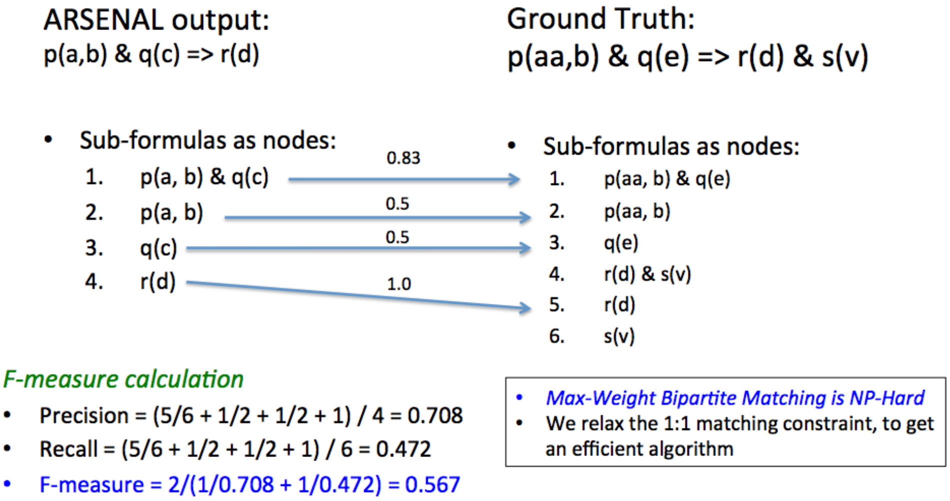
1. LogicalSymbol Token(e.g., "and", "or", "exists"): c(A,B)=1 if A ! = B.

2. String Token (e.g., argument name): c(A,B)=LevenshteinStringEditDistance(A,B).
3. Variable Token (e.g., function name): c(A,B)=0, since function names can change.

Max-weighted matching in bipartite graph We next form a bipartite graph, where the sub-formulas of G and A are the nodes, and the edge connecting each node in G to each node in A is the Typed Levenshtein distance between those 2 formulas.

Max-Weight Biparte Matching is NP-Hard — so, we relax the 1:1 matching constraint in max-weight bipartite matching to get an efficient algorithm. Once we get the matching, we use that to compute the precision, recall and F-measure of the matching.

Figure 18 shows the bipartite graph for an example pair of formulas after matching, where the nodes correspond to the sub-formulas and the edges correspond to the Typed Levenshtein distance between the best matched sub-formulas. The figure also shows the F-measure calculation for this graph.



ARSENAL output:
p(a,b) & q(c) => r(d)

- Sub-formulas as nodes:
  1. p(a, b) & q(c)
  2. p(a, b)
  3. q(c)
  4. r(d)

Ground Truth:
p(aa,b) & q(e) => r(d) & s(v)

- Sub-formulas as nodes:
  1. p(aa, b) & q(e)
  2. p(aa, b)
  3. q(e)
  4. r(d) & s(v)
  5. r(d)
  6. s(v)

F-measure calculation

- Precision = (5/6 + 1/2 + 1/2 + 1) / 4 = 0.708
- Recall = (5/6 + 1/2 + 1/2 + 1) / 6 = 0.472
- F-measure = 2/(1/0.708 + 1/0.472) = 0.567

- *Max-Weight Bipartite Matching is NP-Hard*
- We relax the 1:1 matching constraint, to get an efficient algorithm

Figure 18 Bipartite graph and F-measure calculation corresponding to example formula and ARSENAL output.

Evaluation on Test set We evaluate the F-measure on test sets, which was comprised of other sections in the TTEthernet Requirements document, parts of the TTEthernet standards document, and Eurail requirements document. Table 7 shows the results — overall, on 83 requirements, the NLP stage had an F-measure of 0.62.

7.2 FM Stage: Evaluation

Verification In this section, we demonstrate the effectiveness of the FM stage of the ARSENAL pipeline in creating a full formal model from NL requirements, through the FAA-Isolette example (described earlier in the detailed case study) as well as another real-world example, which has requirements taken from the actual industrial requirements document used by TTTech for their standardized TTEthernet architecture.

Both the SAL model and theorems are automatically generated by ARSENAL from their NL descriptions. In this section, we demonstrate the usefulness of incorporating verification technologies in the ARSENAL pipeline to identify problems in NL documents.

Table 7 NLP stage accuracy on test set.

Section	NumRequirements	F-measure
TTE 3	12	0.59
TTE 7.1	12	0.63
TTE 8.2	7	0.66
TTE 9.1	13	0.67
TTE 9.3	10	0.64
TTE Switch	12	0.65
TTE Standards	6	0.57
Eurail	11	0.54
Total	83	0.62

TTEthernet. In the TTEthernet corpus, we consider the NL requirements that describe the synchronization state machine in TTEthernet. Figure 19 shows the diagram of this state machine (conditions for transitions are not shown). The machine starts at the `ES_INTEGRATE` state, and the `ES_SYNC_ABS` state indicates that the end-system has synchronized with other systems in the cluster.

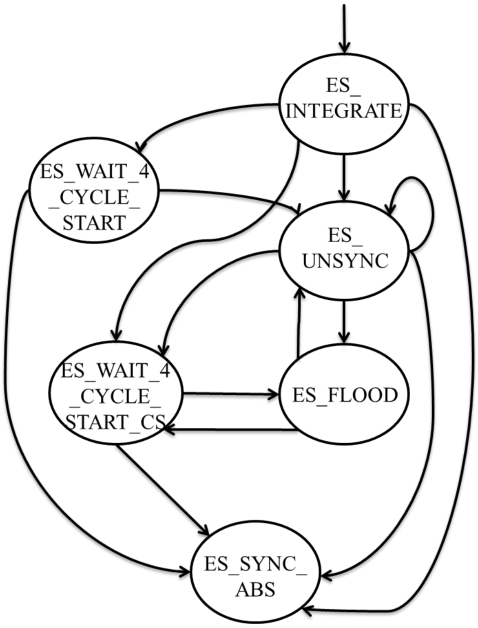


Figure 19 Synchronization FSM in TTEthernet

This corpus contains 36 sentences.[6] ARSENAL can handle complex requirements sentences, generating the correct formula automatically. An example, describing part of the behavior in the `ES_UNSYNC` state, is shown below.

REQ3: *When an end system is in ES_UNSYNC state and receives a coldstart frame, it shall (a) transit to ES_FLOOD state, (b) set local_timer to es_cs_offset, (c) set local_clock to* 0*, (d) set local_integration_cycle to*

[6] The requirements corpora for the FAA-Isolette and TTEthernet domains, and the corresponding SAL models generated by ARSENAL, are available at: `http://www.csl.sri.com/~shalini/arsenal/`.

0*, and (e) set local_membership_comp to* 0*.*

Note that this sentence has a more complicated structure than REQ1 and includes five itemized actions. From the overall SAL model generated automatically, the part corresponding to REQ3 is shown in Figure 20. Observe that ARSENAL was able to infer that the end-system has an enumerated type (Type0) which contains named values `ES_UNSYNC_state` and `ES_FLOOD_state`. It was also able to set correctly the type of `local_integration_cycle` and `local_membership_comp` to INTEGER. In this example, the user asserted that all the five LOCAL variables are *state* variables. Hence, the actions over these variables were considered as state updates and mapped to the TRANSITION section. The formula generated by the SAL adapter corresponding to REQ3 is therefore placed in this section of the SAL model.

```
tte_example : CONTEXT =
BEGIN
  Type1 : TYPE = {coldstart_frame};
  Type0 : TYPE = {ES_UNSYNC_state, ES_FLOOD_state};
  Type2 : TYPE = {es_cs_offset};
  main : MODULE =
  BEGIN
    LOCAL local_integration_cycle : INTEGER
    LOCAL local_membership_comp : INTEGER
    LOCAL local_clock : INTEGER
    LOCAL end_system : Type0
    LOCAL local_timer : Type2
    INPUT in_channel : Type1
    TRANSITION
      [ (end_system = ES_UNSYNC_state AND
          in_channel = coldstart_frame) -->
          end_system' = ES_FLOOD_state;
          local_timer' = es_cs_offset;
          local_clock' = 0;
          local_integration_cycle' = 0;
          local_membership_comp' = 0 ]
  END;
END
```

Figure 20 SAL Model for REQ3.

A formal method expert was asked to review the model and found it was compatible with (and in fact, included more information than) a similar model that he handcrafted in [SD10]. We then asked one of the original creators of the TTEthernet documentation to provide a high-level specification that should be verified for this model. The sentence in English is given below, followed by the corresponding LTL theorem in SAL syntax generated by ARSENAL.

REQ4: *If the end system is in ES_FLOOD state, it shall eventually not be in ES_FLOOD state.*

```
THEOREM main |- G((end_system = ES_FLOOD_state =>
             F(NOT(end_system = ES_FLOOD_state))));
```

We applied bounded model checking, a model checking technique that checks if the model satisfies the requirement within a bounded number of transitions, and found a counterexample. This counterexample reveals that if the environment keeps sending a `coldstart_frame` to this module, then `local_timer`, which maintains a count to timeout in the `ES_FLOOD_state`, will keep resetting to 0 and thus preventing any transition out of the `ES_FLOOD_state` to occur. This helped us identify the missing assumption (absent in the original documentation) that was needed for system verification. In fact, modular verification is one of the most difficult tasks in verification since it requires the precise specifications of the constraints on the environment. These constraints are often implicit and undocumented. In this case, the interaction of multiple end-systems should ensure that any end-system will not receive a `coldstart_frame` infinitely often before it can exit the `ES_FLOOD_state`.

FAA-Isolette. Figure 21 (a) shows one of the finite state machines corresponding to the regulator function in the FAA document. In addition to NL requirements specified in this document, NL sentences were manually written for each transition in the FSMs (including the one shown here). An example sentence is shown below.

REQ5: *If the Regulator Mode equals INIT and the Regulator Status equals True, the Regulator Mode shall be set to NORMAL.*

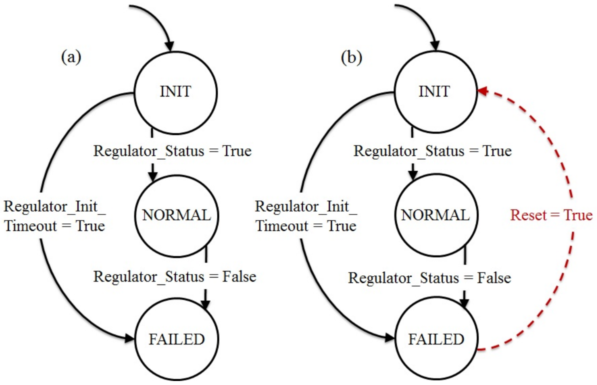


Figure 21 Original FSM (a) and Modified FSM (b) for Regulator.

This experiment seeks to evaluate if ARSENAL can faithfully generate the transition system corresponding to the description (including the FSM) in the design document. Similar to the analysis performed for the TTEthernet example, verification was used to validate the generated FAA model. The corresponding SAL theorem generated by ARSENAL is shown below.

```
THEOREM main |- G((Regulator_Mode = FAILED =>
          NOT (F(Regulator_Mode = NORMAL))));
```

This theorem states that if the FSM is in the FAILED state, then it cannot go back to the NORMAL state (`F` in the theorem means *eventually*). Applying model checking, we verified that the generated SAL model satisfied the theorem. In general, for systems with a large state space like the TTEthernet example, it would be difficult to prove such theorems by manual inspection alone.

To demonstrate the applicability of ARSENAL in identifying inconsistencies in NL requirements, we added a sentence corresponding to the transition from the FAILED state to the INIT state, as shown in Figure 21 (b). For the modified model, ARSENAL quickly produced a counterexample that showed a path from the FAILED state to the NORMAL state, thus violating the aforementioned theorem. This demonstrates that the SAL model generated automatically by ARSENAL behaves as expected. By integrating an NLP pipeline with formal analysis engines, ARSENAL can bring significant benefits to the requirements engineering community by detecting problems in NL requirements.

Synthesis We further applied the LTL realizability and synthesis analysis to the FAA-Isolette corpus. In this scenario, each sentence in the corpus is interpreted by ARSENAL as an end-to-end requirement on the target implementation. Hence, any non-input variable is considered as an output. Additionally, in order to work with the GR(1) synthesis tool RATSY, all the variables are converted to the bit-level.

Application of LTL [GR(1)] synthesis to these formulas produced an unrealizable result; no Moore machine existed to satisfy the formulas. At this point, the user can either interact with the tool (RATSY) to debug the specification or directly examine candidate assumptions generated by ARSENAL. The latter is more user-friendly since it assumes no knowledge of formal methods and other tools on the part of the user, and enables a user to directly refine the existing NL requirements. For the FAA-Isolette example, ARSENAL produces the following candidate assumption to make the specification realizable.

```
G !(Regulator_Status=1 & Regulator_Init_Timeout=1);
```

To better understand why this assumption is necessary for ARSENAL to generate a SAL model from the set of sentences, observe that in Figure 21, the INIT state has two outgoing transitions, one to the NORMAL state and the other to the FAILED state. When both `Regulator_Status` and `Regulator_Init_Timeout` are `true` , the state machine can nondeterministically choose to go to either of the states. Such behavior is not desirable in an actual implementation, which is supposed to be deterministic. Hence, the original specification is not realizable.

In this example, if the NL sentences describing the transitions were written differently, in a way that `Regulator_Status`=1 and `Regulator_Init_Timeout`=1 were mutually exclusive, then the specification would be realizable and an implementation could also be generated automatically in Verilog. In general, the notion of unrealizabilty captures a much wider class of bugs than nondeterminism, and assumption mining helps to generate candidate fixes and facilitates interaction, especially with end-users.

8 Related Work

We first give an overview of the research related to ARSENAL.

8.1 Summary of Related Work

There is a rich and diverse body of research related to requirements engineering. The main advantages of ARSENAL over prior work are a less restrictive NL front-end, a more powerful FM analysis framework, and a stronger interaction between the NL and FM stages.

Kress-Gazit et al. [KGFP08], Smith et al. [SACO02] and Shimizu et al. [Shi02] propose grammars for representing requirements in controlled natural language. The natural language interfaces suggested in these papers are restrictive — ARSENAL can extract information automatically from a wider range of natural language styles.

Zowghi et al. [ZGM01], Gervasi et al. [GZ05], Scott et al. [SCK04], Xiao et al. [XPTX12], and Ding et al. [DJP11] process constrained natural language text using NLP tools (e.g., CFG, Cico) and perform different types of checks (e.g., consistency, access control) on the requirements. Compared to these methods, ARSENAL uses more state-of-the-art NLP techniques that can be made domain-specific using resources like the domain-specific ontology, customized regex-based template matching in pre-processing, etc. ARSENAL also uses more advanced model checking tools (e.g., SAL), which can represent theorems and invariants in a much more expressive logic (e.g., LTL).

Behavior-Driven Development (BDD) is a way to give natural language specifications for the software testing phase. Drechsler et al. [DDG+12], Soeken et al. [SWD12], and Harris [Har12] show different ways of translating high-level natural language requirements to tests in the BDD framework, which can then be used by BDD tools like Cucumber [cuc] or RSpec [rsp]. ARSENAL is more general than these approaches — instead of considering requirements specifications at the test phase, it considers NL requirement specifications that can be specified and verified in the design phase.

Ormandjieva et al. [OKH07], QUARS [FFGL01], and Goguen [Gog96] focus on assessing the quality of requirements documents — they do not create formal models from the requirements for downstream formal analysis (e.g., consistency checks) like ARSENAL. Malin et al. [Mal09], Boyd [Boy99], and Nikora et al. [NB09] do linguistic modeling and information extraction from requirements documents, but do not handle consistency checks or downstream formal methods analysis (e.g., using SAL) like ARSENAL. Attempto Controlled English (ACE) [SF96], RECORD [Bor96] and T-RED [BS96] are user-oriented tools for requirements collection, reuse, and documentation. These are interactive tools requiring inputs from domain experts, and are not as automated as ARSENAL.

8.2 Details of Related Work

We next discuss details of related research in two major areas relevant to our work — requirements engineering and natural language processing — and highlight the comparative advantage of ARSENAL.

Requirements Engineering We begin with a comparison of ARSENAL to two approaches in requirements document analysis.

PROPEL: In PROPEL [SACO02], the authors note that properties used in formal verification map to one of several property pattern templates. These pattern templates are shown to the user in an interactive framework as choosing among these options should help the specifier consider the relevant alternatives and subtleties associated with the intended behavior.

The authors represent these pattern templates using two different notations: an extended Finite State Automaton (FSA) representation and a Disciplined Natural Language (DNL) representation. The DNL representation provides a short list of alternative phrases that highlight the options, as well as synonyms for each option to support customization. This representation should appeal to those specifiers who prefer a natural language description, and do not want to handle the full rigor of formal property specifications. The extended FSA representation provides a graphical view that can be used to derive a specific FSA representation.

This work differs with our approach in a few important ways:

1. The authors do not extract property templates automatically from given natural language specifications. Instead, they provide a tool that facilitates the user in writing specifications, and gives tips to modify the specification to make it less ambiguous. ARSENAL can extract information automatically from given natural language specifications, and does not have to rely solely on the user to modify specifications like PROPEL does.
2. ARSENAL handles more general input sources, e.g., specs from tables, flow charts.
3. ARSENAL also has a mechanism for checking consistency across multiple specifications, which is a generalization of PROPEL.

CARL: Zowghi et al. [ZGM01] address the issue of finding inconsistencies in requirement documents. They represent requirements as constrained natural language text and convert these specifications into formulas in propositional logic using an NLP parser, Cico. Then they use a reasoning system, CARET, to analyze the inconsistency in specifications, where requirements are modeled as default theories. The authors demonstrate the effectiveness of their approach using data from the London Ambulance System (LAS), which uses a computer-aided dispatch system. They evaluate this case study in detail and find inconsistencies in particular requirements specifications in LAS.

Some of the key advantages of ARSENAL over CARL are:

1. CARL uses a simple NLP tool called Cico, which parses the text to get entities and relations that are used to get the formulas directly. ARSENAL uses more state-of-the-art NLP techniques, which can be made domain-specific using resources like the glossary, customized regex-based template matching, and so on.
2. The consistency engine CARET uses classical logic and non-monotonic reasoning. The underlying logic used by the formal methods approaches in CARL is default theory, which uses deterministic propositional formulas. In comparison, ARSENAL uses more advanced model checking tools (e.g., SAL).

Other Research

Some other notable related research activities in requirements engineering are discussed next, each of which has technology related to some parts of the ARSENAL pipeline.

1. Gervasi et al. [GZ05] explore the integration of natural language parsing techniques with default reasoning to overcome the difficulty of training system designers directly using logic. They also propose a method for automatically discovering inconsistencies in the requirements from multiple sources, using both theorem-proving and model-checking techniques. The effectiveness of their approach and tool is illustrated on an example domain involving conflicting requirements. Scott et al. [SCK04] have proposed a context-free grammar to facilitate the parsing of requirement text, e.g., finding temporal clauses, condition clauses, relative clauses. The grammar is used to drive an interactive tool that takes the specification input by a user and standardizes it to the predefined format specified by the grammar, prompting the user to accept the re-formatting. The structured requirements can then be used by case-based reasoning systems for comparison and consistency checks.

   While the above systems had goals similar to those of ARSENAL, the latter uses more advanced and state-of-the-art NLP, machine learning and formal methods tools and can therefore handle more complex (potentially ambiguous) specifications of different types (from multiple sources).
2. Ormandjieva et al. [OKH07] propose a quality model for evaluating requirements text, and a text classification system to automate the quality assessment process. The text classifier was trained on features that were extracted automatically from the text (output of a Part-of-Speech tagger and a parser), with human judgment as ground truth. The classifier was able to actually flag ambiguous and unambiguous texts at the surface level of understanding. They ran a large study using human raters. Inter-rater agreement levels showed that the quality assessment task is difficult for humans, but there is reasonable agreement on the chosen quality indicators, both at the level of surface understanding and at the level of conceptual understanding. The study also demonstrated that automatic detection of ambiguities in requirements documentation has good performance, comparable to human judgment. Other relevant work in this area is QUARS by Firing et al. [FFGL01], a tool for automatic quality evaluation for natural language requirement. Goguen [Gog96] also discusses the trade-offs between formal representation and informal specifications in requirements documents.

   The above papers focus only on assessing the quality of requirements documents or predicting the judgment of a human user about the quality of the requirements. In contrast, ARSENAL is much broader in scope.
3. The Automated Tool and Method for System Safety Analysis project [Mal09] used a Semantic Text Analysis Tool to extract key information from Failure Models and Effects Analysis (FMEA) and hazard reports. The primary sources of extractions are FMEA documents, which contain system descriptions, problem descriptions, and statements about connections and dependencies. Model generation software in the Hazard Identification Tool integrates this information into visualizations of system architecture models. The intent is to make it easier to review hazard paths and find redundant and missing links within and between types of analysis. Boyd [Boy99] describes a controlled natural language that can be used to specify software development models.

   The above papers do linguistic modeling and information extraction from requirements documents, but do not handle consistency checks or downstream formal methods analysis like ARSENAL.
4. Crow et al. [CV96] use formal methods tools (PVS, Murphy) for case studies on the flight-software subsystem in the NASA shuttle program, using formal specification techniques or using state exploration approaches. The key technical results of the paper are a clear demonstration of the utility of formal methods, not as a replacement but as a complement to the conventional Shuttle requirements analysis process. The application of formal methods to the particular projects considered in this paper – JS, 3E/O, and GPS – each uncovered anomalies ranging from minor to substantive, most of which were undetected by existing requirements analysis processes. The main insight from the paper is that formal methods techniques are most effective when they are judiciously tailored to the application.

   This work does not have an end-to-end system involving information extraction, consistency checking and formal verification like ARSENAL – it focuses only on the formal methods part.
5. Another domain-specific language is ACE [SF96], a subset of English with a restricted grammar and a domain-specific vocabulary, which allows domain specialists to interactively formulate requirements specifications in domain concepts. RECORD [Bor96] and T-RED [BS96] are user-oriented tools for requirements collection, reuse, and documentation.

These are interactive tools requiring inputs from domain experts and are not fully automated like ARSENAL.
6. Shimizu et al. [Shi02] discuss how formal specification can be written from natural language requirements, for commonly used interface protocols, and how test inputs and the checking properties can be generated automatically from the formal specification.
However, unlike ARSENAL, the above work does not involve automatic extraction of formal specifications from natural language.

Natural Language Processing (NLP) Here we discuss related work corresponding to the NLP part of the ARSENAL pipeline, which does information extraction from semi-structured text. This problem has been studied in different domains, e.g., clinical text [dBCK$^+$11], legal documents [SNM10], web tables [CP10]. For example, the authors in [SNM10] have an approach where only relevant text segments (e.g., corresponding to litigation claims) are extracted from a full legal document, and then relevant entities (e.g., patents, laws) are extracted from those text segments. In ARSENAL we extract *both* entities and relations using NLP tools, and additionally connect their output to Formal Methods tools to facilitate more detailed downstream analysis.

Within the NLP portion of the ARSENAL pipeline, another core component is the extraction of text spans to fill the slots in the template structure. The following related work also use the general idea of breaking documents into text spans for further processing:

- Teufel et al. [TM02] present an approach to summarizing scientific articles that is based on the idea of restoring the discourse context of extracted material by adding the rhetorical status to each sentence in a document. The innovation of their approach is that it defines principles for content selection specifically for scientific articles, and that it combines sentence extraction with robust discourse analysis. The output of their system is a list of extracted sentences along with their rhetorical status (one of seven rhetorical categories). The output of the proposed extraction and classification system can be viewed as a single-document summary. In addition, it provides starting material for the generation of task-oriented and user-tailored summaries designed to give users an overview of a scientific field. The authors present several experiments measuring the agreement of human judges with the system output, on the rhetorical annotations.
- Heilman et al. [HS10] present an algorithm for extracting simplified declarative sentences from syntactically complex sentences. They motivate their extraction approach by issues that are relevant for automatic question generation. The authors extract simplified sentences from complex linguistic structures in documents, e.g., appositives, subordinate clauses. They use the extracted simple directives to automatically generate questions based on the input text. Their system successfully simplifies sentences and transforms them into questions. They evaluate the approach using experiments and show that it is more suitable for extraction of factual question generation than a standard text compression baseline, which takes as input a possibly long and complex sentence and produces as output a single shortened version to convey the main piece of information in the input.
- Barker et al. [BCC$^+$04] present a question-answering system that was developed as part of the HALO project, and discuss the results of its evaluation. The system was developed using a combination of several knowledge representation and reasoning technologies, in particular semantically well-defined frame systems, automatic classification methods, reusable ontologies, and a methodology for knowledge base construction. The system was able to encode the knowledge from 70 pages of a college-level chemistry textbook into a declarative knowledge base, and successfully answer questions comparable to questions on an Advanced Placement exam with high accuracy. In addition, the authors extended existing explanation generation methods, allowing the system to produce high-quality English explanations of its reasoning. The resulting system, in addition to having high accuracy, gave explanations that are comparable in quality to human judgments on tests.

In ARSENAL, since we have a controlled vocabulary for the domain and the underlying text is semi-structured, we propose to use a regular expression (regex)-based text span extraction technique – we take this simple but flexible approach, so that the regex matching rules can be updated effectively, and learned from data if necessary.

Compliance checking and monitoring There are other compliance checking and monitoring tools in the privacy and security domains. The HIPAA Compliance Checker [BDMN06] is a formal translation of HIPAA into Prolog – the current implementation checks compliance with HIPAA rules, doing goal-driven evaluation of privacy policies and developing automated support for privacy policy compliance and audit. Basin et al. [BKM10] monitor complex security properties using a runtime approach using metric first-order temporal logic, and experimentally evaluate the performance of the resulting monitors. These approaches do not generate the logic formulas directly from the natural language specifications, which is the task that ARSENAL focuses on.

9 Conclusions

The key accomplishments of ARSENAL are outlined in Figure 22.

1. **Creating an NLP workflow for generating the IR**:
   - a) ARSENAL does semantic parsing using the combination of a type dependency parser, metatags and type rules,
   - b) Resolves co-references and ambiguities in complex requirements sentences,
   - c) Handles both domain-independent (e.g, for arithmetic expressions) and domain-specific pre-processing.
2. **Creating a FM workflow to generate a complete SAL model from the IR**:
   - a) ARSENAL has multiple output generators, to generate the appropriate output (e.g., FOL formula, SAL model) for a domain.
   - b) For SAL model generation from IR, ARSENAL
     - (i) Has principles to determine which formula should go to which part of the SAL model automatically.
     - (ii) Automatically determines the SAL types, when the user only provides the input type categories.
     - (iii) Guides the user to come up with the right formulation of the FM theorem, in natural language.
     - (iv) Provides a debugging environment to the FM expert, helping to discover missing assumptions in the text.
   - c) ARSENAL generates counter-examples, constructs proofs of properties, and uses realizability to check inconsistency of requirements.
3. **Connecting the NLP and FM stages to create an end-to-end pipeline for both FAA-Isolette and TTEthernet domains**:
   - a) ARSENAL was developed on the FAA domain and later ported to the more complex TTEthernet domain,
   - b) Has a modular design that helped isolate the parts that needed to be changed (e.g., pre-processor) without modifying the core parts,
   - c) Has many algorithms (e.g., type rules) that are quite robust to porting to a new domain.
4. **Designing novel evaluation metrics to assess the performance of ARSENAL** (detailed numbers in Section 5.1):
   - a) ARSENAL is automated to a large degree (as measured by the degree of automation metric),
   - b) Is robust to requirements perturbation (as measured by the degree of perturbation metric),
   - c) Has good accuracy in the NLP stage in generating the output formula on test datasets (as measured by a novel accuracy metric).
5. **Saving significant development cycles of the end-user**:
   - a) ARSENAL creates a first-cut formal model automatically from voluminous requirements, manually creating which requires a significant effort,
   - b) Needs the user input to be provided only once per application domain,
   - c) Allows user training efforts in formal modeling to be minimized.

Figure 22 Key accomplishments in ARSENAL.

Some of the benefits of ARSENAL include: (1) resolution of semantic ambiguities and co-references in requirements, (2) consistency/redundancy checking and validation of requirements, (3) example generation for test cases, (4) putative theorem exploration, (5) traceability to connect implementations to requirements they implement, (6) feedback on requirements quality and hints for improvement to the end user, facilitating iterative requirements refinement. Overall, ARSENAL facilitates communication between stakeholders (e.g., formal methods modelers, requirements engineers), which is important for critical systems like avionics, and helps to refine imprecise requirements.

In the future, we would place primary emphasis on making the ARSENAL framework more robust. We want to test ARSENAL on multiple other domains and datasets, and design more evaluation metrics like the ones discussed in this paper (e.g., automation and perturbation metrics) to evaluate the performance of the ARSENAL pipeline as we improve it. We would also like to create benchmark datasets for evaluating different aspects of ARSENAL. Apart from SAL models, we have also experimented with other logical model outputs, e.g., first-order logic. We plan to continue generating other logical models, which could be suitable for other types of formal analysis. We would also like to explore the creation of richer system models, by composing models generated from separate requirements corpora. The plug-and-play ARSENAL architecture will also facilitate trying out NL parsers other than STDP in the future.

The current ARSENAL system also has a statistics generator, which generates statistics about the distribution of entities, typed dependencies, etc. in a requirements corpus. We use the generator to identify important type rules (e.g., from dominant TDs) and important preprocessing rules (e.g., from dominant entities) for ARSENAL. We would like to use these statistics and apply machine learning to automatically customize different parts of ARSENAL (e.g., type rules, translation rules) for a given domain and requirements corpus.

So far we have only considered requirements in natural language text. In the future, we would also like to parse flow-charts, diagrams and unstructured tables in requirements, as well as handle events, intervals, and other complex NL constructs. We would also like to generalize the ARSENAL pipeline beyond formal analysis to generate probabilistic models, in domains where probabilistic analysis is useful (e.g., for failure analysis using probabilistic models [GSDL13]).

10 Acknowledgments

Special thanks to Kathleen Fisher, Bruno Dutertre, Sam Owre, John Rushby, Ashish Tiwari, and Brendan Hall for all their help and guidance regarding this work. This material is based upon work supported by the United States (US) Air Force and the Defense Advanced Research Projects Agency (DARPA) under Contract Numbers FA8750-12-C-0339 and FA8750-12-C-0284. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US Air Force and DARPA.

References

Bab07. J. Babcock. Good requirements are more than just accurate. Practical Analyst: Practical Insight for Business Analysts and Project Professionals, December 2007.

BCC$^+$04. K. Barker, V. Chaudhri, S.Y. Chaw, P.E. Clark, J. Fan, D. Israel, S. Mishra, B. Porter, P. Romero, D. Tecuci, and P. Yeh. A question answering system for AP chemistry: Assessing KR technologies. In *Proceedings of International Conference on Knowledge Representation and Reasoning*, 2004.

BCG$^+$10. Roderick Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Konighofer, Marco Roveri, Viktor Schuppan, and Richard Seeber. Ratsy: A new requirements analysis tool with synthesis. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 425–429. Springer Berlin Heidelberg, 2010.

BDMN06. Adam Barth, Anupam Datta, John C. Mitchell, and Helen Nissenbaum. Privacy and contextual integrity: Framework and applications. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 184–198, 2006.

BGJ$^+$07. R. Bloem, S. Galler, B. Jobstmann, N. Piterman, Amir Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: A case study. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, 2007.

BGL$^+$00. Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Mu noz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, jun 2000. NASA Langley Research Center.

BKM10. David Basin, Felix Klaedtke, and Samuel Müller. Monitoring security policies with metric first-order temporal logic. In *Proceedings of the 15th ACM Symposium on Access control models and technologies*, pages 23–34, 2010.

Bor96. J. Borstler. User-centered requirements engineering in record - an overview. In *Proceedings of Nordic Workshop on Programming Environment Research (NWPER)*, 1996.

Boy99. N. Boyd. Using natural language in software development. *Journal of Object Oriented Programming*, 11(9), 1999.

BP98. Barry W. Boehm and Philip N. Papaccio. Understanding and controlling software costs. *IEEE Transactions on Software Engineering*, 14(10):1462–1477, October 1998.

BS96. Tomas Boman and Katarina Sigerud. Requirements elicitation and documentation using T-red. Master's thesis, Univeversity of Umeå, 1996.

bug10. Software defects - do late bugs really cost more? Slashdot Article (http://tinyurl.com/8vew93k), March 2010.

CBRZ01. Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Form. Methods Syst. Des.*, 19(1):7–34, July 2001.

CGP99. Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model checking.* MIT Press, Cambridge, MA, USA, 1999.

CP10. Eric Crestan and Patrick Pantel. Web-scale knowledge extraction from semi-structured tables. In *Proceedings of International Conference on World Wide Web (WWW)*, 2010.

cuc. Cucumber. http://cukes.info.

CV96. Judith Crow and Ben L. Di Vit. Formalizing space shuttle software requirements. In *ACM SIGSOFT Workshop on Formal Methods in Software Practice*, 1996.

dBCK$^+$11. Berry de Bruijn, Colin Cherry, Svetlana Kiritchenko, Joel D. Martin, and Xiaodan Zhu. Machine-learned solutions for three stages of clinical information extraction: The state of the art at i2b2 2010. *JAMIA*, 2011.

DDG$^+$12. Rolf Drechsler, Melanie Diepenbeck, Daniel Große, Ulrich Kühne, Hoang M. Le, Julia Seiter, Mathias Soeken, and Robert Wille. Completeness-driven development. In *International Conference on Graph Transformation*, 2012.

DJP11. Z. Ding, M. Jiang, and J. Palsberg. From textual use cases to service component models. In *Proceedings of 3rd International Workshop on Principles of Engineering Service-Oriented Systems*, pages 8–14, 2011.

dMMM06. Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. Generating typed dependency parses from phrase structure parses. In *In Proc. Intl. Conf. on language resources and evaluation (LREC)*, pages 449–454, 2006.

FFGL01. F. Fabbrini, M. Fusani, S. Gnesi, and G. Lami. An automatic quality evaluation for natural language requirements. In *Proceedings of International Workshop on RE: Foundation for Software Quality*, 2001.

GJ01. Daniel Gildea and Daniel Jurafsky. Automatic labeling of semantic roles. *Computational Linguistics*, 28:245–288, 2001.

GLM08. Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2008.

Gog96. Joseph A. Goguen. Formality and informality in requirements engineering. In *Proceedings of International Conference on Requirements Engineering*, 1996.

GSDL13. Shalini Ghosh, Wilfried Steiner, Grit Denker, and Patrick Lincoln. Probabilistic modeling of failure dependencies using Markov logic networks. In *Proceedings of the 19th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2013.

GZ05. Vincenzo Gervasi and Didar Zowghi. Reasoning about inconsistencies in natural language requirements. *ACM Trans. Softw. Eng. Methodol.*, 14, July 2005.

Har12. Ian G. Harris. Extracting design information from natural language specifications. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1256–1257, 2012.

HS10. Michael Heilman and Noah A. Smith. Extracting simplified statements for factual question generation. In *Proceedings of AIED Workshop on Question Generation*, 2010.

KGFP08. Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. Translating structured English to robot controllers. *Advanced Robotics*, pages 1343–1359, 2008.

Kup12. Orna Kupferman. Recent challenges and ideas in temporal synthesis. In *SOFSEM 2012: Theory and Practice of Computer Science*, volume 7147 of *Lecture Notes in Computer Science*, pages 88–98. Springer Berlin Heidelberg, 2012.

LDS11. Wenchao Li, L. Dworkin, and S.A. Seshia. Mining assumptions for synthesis. In *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 43–50, 2011.

LM09a. David L. Lempia and Steven P. Miller. Requirements engineering management handbook. Final Report DOT/FAA/AR-08/32, Federal Aviation Administration, June 2009.

LM09b. David L. Lempia and Steven P. Miller. Requirements engineering management handbook. Final Report DOT/FAA/AR-08/32, Federal Aviation Administration, June 2009.

Mal09. Jane T. Malin. Automated tool and method for system safety analysis: 2009 progress report. Technical Report NASA/TM-2010-214800, NASA, 2009.

Mil95. George A. Miller. Wordnet: A lexical database for English. *Communications of the ACM*, 38:39–41, 1995.

MP92. Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. 1992.

NB09. A.P. Nikora and G. Balcom. Automated identification of LTL patterns in natural language requirements. In *20th International Symposium on Software Reliability Engineering (ISSRE)*, 2009.

OKH07. Olga Ormandjieva, Leila Kosseim, and Ishrar Hussain. Toward a text classification system for the quality assessment of software requirements written in natural language. In *European Conference on Software Quality Assurance*, 2007.

ORR$^+$96. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

PP06. Nir Piterman and Amir Pnueli. Synthesis of reactive(1) designs. In *In Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 364–380, 2006.

rsp. Rspec. http://en.wikipedia.org/wiki/RSpec.

RV07. Kristin. Rozier and Moshe. Vardi. LTL satisfiability checking. In *Model Checking Software*, volume 4595 of *Lecture Notes in Computer Science*, pages 149–167. Springer Berlin Heidelberg, 2007.

SACO02. Rachel L. Smith, George S. Avrunin, Lori A. Clarke, and Leon J. Osterweil. PROPEL: An approach supporting property elucidation. In *24th International Conference on Software Engineering*, 2002.

SCK04. William Scott, Stephen Cook, and Joseph Kasser. Development and application of context-free grammar for requirements. In *System Engineering Test and Evaluation Conference (SETE)*, 2004.

SD10. W. Steiner and B. Dutertre. SMT-based formal verification of a TTEthernet synchronization function. In *FMICS*, 2010.

SF96. Rolf Schwitter and Norbert E. Fuchs. Attempto controlled English (ACE) a seemingly informal bridgehead in formal territory. In *JICSLP*, 1996.

Shi02. Kanna Shimizu. *Writing, Verifying, and Exploiting Formal Specifications for Hardware Designs*. PhD thesis, Department of Electrical Engineering, Stanford University, August 2002.

SNM10. Mihai Surdeanu, Ramesh Nallapati, and Christopher Manning. Legal claim identification: Information extraction with hierarchically labeled data. In *Semantic Processing of Legal Texts Workshop*, 2010.

SWD12. Mathias Soeken, Robert Wille, and Rolf Drechsler. Assisted behavior driven development using natural language processing. In *Objects, Models, Components, Patterns*, volume 7304 of *Lecture Notes in Computer Science*, pages 269–287. 2012.

TM02. S. Teufel and M. Moens. Summarizing scientic articles: Experiments with relevance and rhetorical status. *Computational Linguistics*, 28(4), 2002.

XPTX12. Xusheng Xiao, Amit M. Paradkar, Suresh Thummalapenta, and Tao Xie. Automated extraction of security policies from natural-language software documents. In *SIGSOFT FSE*, page 12, 2012.

ZGM01. Didar Zowghi, Vincenzo Gervasi, and Andrew McRae. Using default reasoning to discover inconsistencies in natural language requirements. In *Asia-Pacific Software Engineering Conference (APSEC)*, 2001.

Acronyms

ACE Attempto Controlled English.
ARSENAL Automatic Requirements Specification Extraction from Natural Language.

BDD Behavior-Driven Development.

DFS Depth-First Search.
DNL Disciplined Natural Language.

FAA Federal Aviation Administration.
FM Formal Methods.
FMEA Failure Models and Effects Analysis.
FOL First-Order Logic.
FSA Finite State Automaton.
FSM Finite State Machine.
FTR Final Technical Report.

GR(1) Generalized Reactivity (1).

IR Intermediate Representation.

LAS London Ambulance System.
LHS left-hand side.
LTL Linear Temporal Logic.

MLN Markov Logic Network.

NICU neonatal intensive care unit.
NL Natural Language.
NLP Natural Language Processing.

PVS Prototype Verification System.

RATSY Requirements Analysis Tool with Synthesis.
RHS right-hand side.

SAL Symbolic Analysis Laboratory.
STDP Stanford Type Dependency Parser.

TD typed dependency.
TTEthernet Time-Triggered Ethernet.

Glossary

Büchi automata is a type of ω-automata for infinite words.

isolette is an incubator for infants that provides controlled temperature humidity and an oxygen supply.

model checking refers to the problem of checking whether a model of a system meets a given specification.

RATSY is a requirement analysis tool developed by the Embedded System Unit of FBK and the Institute for Applied Information Processing and Communications of Graz University of Technology. It has the capability to synthesize reactive systems form their temporal specifications.

SAL is a language for specifying concurrent systems in a compositional way, with the support of state-of-the-art model checkers. It is developed at the Computer Science Laboratory of SRI International in Menlo Park, California, in collaboration with Stanford and UC Berkeley.